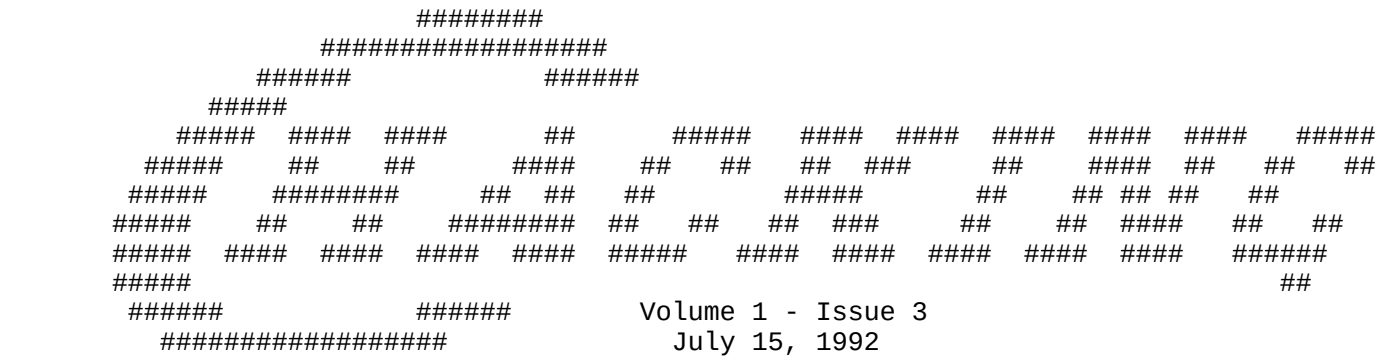


Volume 1 - Issue 3

July 15, 1992

==========

Editor's Notes:
by Craig Taylor (duck@pembvax1.pembroke.edu)

Here's over 3000 lines of hacking articles & such... Sorry about the length as some of the articles ran a bit overboard. Included within is a discussion of the KERNAL routines, an examination of Rasters, and a package of burst routines for use in your own programs. If you've got any ideas for articles etc that you'd like to see or want to hear more on a certain topic feel free to email me.

I'm pleased to introduce the Demo Corner where each month we'll report how to achieve some of the graphic and sound effects that are present in many demos that are wondered about.

Note: The article concerning programming and usage of the 1351 mouse has been delayed until the next issue due to time and length constraints.

This file is available via anonymous ftp at tybalt.caltech.edu under pub/rknop/hacking.mag. Back issues of C= Hacking are also located there.

********** WARNINGS, UPDATES, BUG REPORTS, ETC... **********

OOPS - In the last issue of C= Hacking in Mark Lawrence's File Splitter a line inadvertantly got chopped off. The following code should be fixed between the comments that are listed:

```
[.
  .
   .]
   { Make EXTENSION a string representation of COUNT, to be added to the
      OutFileName to make things a tad easier}

    OutFileName := Concat(NewFile,'.',Copy('00',1,3-Length(Extension)),
                   Extension);   {**THIS IS THE STATEMENT...**}

    { Create filename based on which part we're up to }
[.
  .
   .]
```

==========

Note: Permission is granted to re-distribute this "net-magazine", in whole, freely for non-profit use. However, please contact individual authors for permission to publish or re-distribute articles seperately.

*** AUTHORS LISTED BELOW RETAIN ALL RIGHTS TO THEIR ARTICLES ***

==========

In This Issue:

Learning ML - Part 3

In this edition we take a look at reading and writing commands to the disk drive, including reading the disk directory and error channel. This article parallels the discussion of the C=128 and C=64 KERNAL jump tables of available routines. Written by Craig Taylor.

The Demo Corner: Missing Cycles

Everybody knows that there are 63 cycles available to the C64 processor on each scan line, except for one which only provides 23 cycles. But what happens when we add sprites and why ? Written by Pasi 'Albert' Ojala.

KERNAL 64/128

The C=128 and C=64 jump table points to many valuable system routines is discussed and examined in detail. Written by Craig Taylor.

64K VDC RAM and an alternate GEOS128 Background Screen

Standard GEOS only uses the first 16K of your VDC screen. If you have 64K of VDC RAM, and want to write an 80-column only application, you can put some of the additional VDC RAM to use as a replacement for the standard GEOS background screen. And, in the bargain, you get an additional 16K of application FrontRAM to use! Written by Robert Knop.

GeoPaint File Format

Written by Bruce Vrieling, this article provides an in depth description of exactly how geoPaint stores its graphic images on disk. It examines the concept of VLIR files, how graphics data is laid out on screen (from both geoPaint and the VIC's perspective), and geoPaint's graphics compression techniques.

Rasters - What They Are and How to Use Them

Written by Bruce Vrieling, this article provides an introduction to creating special on-screen effects using the technique of raster interrupts. The basics are examined, including what they are, and how to program them. This article should provide a good starting point for someone wanting to get their feet wet in raster programming.

Bursting Your 128: The Fastload Burst Command

Written by Craig Bruce this article covers the Fastload burst command of the 1571 and 1581 disk drives. The Fastload command operation and protocol are discussed and a package for using the Fastload command to read regular sequential files at binary program loading speeds is presented. To demonstrate the package, a file word counting utility is implemented and the "commented" code is included.

==========

Learning ML - Part 3
by Craig Taylor (duck@pembvax1.pembroke.edu)

Last time we used a routine at $FFD2 which would print out the character code contained within the accumalator. That location will always print the character out regardless of VIC-20, C=64, C=128 and even PET because Commodore decided to set up some locations in high memory that would perform routines that are commonly needed.

Take a look now at the KERNAL 64/128 article and glance over some of the routines and their function / purpose. This article is meant to be a companion to that article so you may want to flip back and forth as the discussion

of the program listed below is discussed.

Note that I've borrowed Craig Bruce's notation of having listings inside. To extract the source that follows enter the following command on a Unix system:

```
grep '^\.@...\!' Hack3 | sed 's/^.@...\!.//' | sed 's/.@...\!//' >dir.asm
```

```
.@001! ;
.@002! ; Set up computer type for computer-dependant code /
.@003! ;    Only used in displaying # routine / start of assembly setting.
.@004! ; BUDDY format.
.@005! ;
.@006! computer = 128             ; Define as either 64 or 128.
```

For both c64 and c128 users the following code works. Within the code is conditional assembly which means it will work on either computer assuming that the computer is equal to either 128 or 64.

```
.@007!
.@008! .if computer-64            ;** if computer not c64 then
.@009!       .org $1300          ;   and also make sure in BANK 15 when calling
.@010!                           ;   these routines.
.@011! .else                      ;** else if _is_ c64, then
.@012!       .org $c000
.@013! .ife                       ;** end of computer-dependant code.
```

Because of this (the source is in BUDDY format) the C64 and C128 are set to assemble at different memory locations. On the C64, $c000 is 49152. On the C128 it is at 4864. Note for the C128 it is necessary to do a BANK15 before executing the code.

```
.@014!       .mem                ; - assemble to memory.
```

This tells the assembler to actually put the code into memory.

```
.@015!
.@016! ;;-------------------------------------------------------------------------
.@017! ;; KERNAL EQUATES
.@018! ;;-------------------------------------------------------------------------
.@019!
.@020! setnam = $ffbd
.@021! setlfs = $ffba
.@022! open   = $ffc0
.@023! close  = $ffc3
.@024! chkin  = $ffc6
.@025! chrin  = $ffcf
.@026! bsout  = $ffd2
.@027! clrch  = $ffcc
.@028!
```

These are the KERNAL routines we will actually be using. Their actual use will be documented when we come across them within the code.

```
.@029! ;;-------------------------------------------------------------------------
.@030!
.@031! temp   = 253
.@032! charret = $0d
.@033! space = $20
.@034!
```

Temp is set up to just be a temporary location in zero-page. Location 253 on both the C64 and C128 is unused. Charret stands for the carriage return character and is the equivlent of a chr$(13). Space stands for the code for a space (a chr$(32))

```
.@035! ;;-----------------------------------------------------------------
.@036!
.@037! start = *
.@038!
.@039!    jsr read'dir       ; Initial jump table -- Note: Will read error after
.@040!    jmp read'err       ;     showing directory.
.@041!
```

  You'll see code like this a lot -- Basically we're building what is known as a jump table. That way if we add more code to the directory or error routine we don't have to worry about our SYS call's changing. To read the directory just SYS base, to read the error channel just SYS base+3 (where BASE is 49152 on the C64, 4864 on the 128)...

  Also the JSR JMP combination may seem a little strange but what we are doing is treating the directory routine as a subroutine and then JUMPING to the error routine. Once we do that the RTS in read'err will return us back to basic.

```
.@042! ;;-----------------------------------------------------------------
.@043!
.@044! read'dir = *
.@045!
.@046! ; Opens and reads directory as a basic program.
.@047! ;==
.@048! ; Basic programs are read in as follows:
.@049! ;                  [Ptr to Next Line]:2 [Line #]:2 [Text]:.... [$00 byte]
.@050! ;                  ^^^^^^^^^^^REPEATS^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.@051! ; The end of a program is signifed by the $00 byte signifying end of text
.@052! ;    and the ptr's also being = $00.
.@053! ;==
```

  There are several ways to read the directory in machine language. What we are doing here is taking advantage of the drive's ability to allow us to load the directory as a basic program -*except*- we aren't loading it per se. We're gonna grab each byte as it comes from the drive and interpret it ourselves instead of putting it in memory as would normally be done.

  Basic programs are stored as the following: A 2 byte pointer, a 2 byte line #, the basic text, and a null terminator and then starting over from the 2 byte pointer. The pointer we do not need, the line # is the number of blocks the file takes up and the TEXT is the program name and file type. We know when we're finished on the line by checking for a $00 byte.

```
.@054!                              ; Begin by opening up the
.@055!                              ; directory file ("$").
.@056!       lda #$01               ;   length is 1
.@057!       ldx #<dir              ;   lo byte pointer to file name.
.@058!       ldy #>dir              ;   hi byte pointer to file name.
.@059!       jsr setnam             ; - call setnam
```

  Okay, first we need to simulate opening the directory as a program file. SETNAM sets up the filename for the open command. In effect we are giving the basic syntax of open file#,device#,channel#,"filename" in reverse.

```
.@060!       lda #$01               ;   file # 1
.@061!       ldx #$08               ;   device # 8
.@062!       ldy #$00               ;   channel # 0
.@063!       jsr setlfs             ; - call setlfs
```

  Here we specify the device #, file #, channel # in preperation for the open.

```
.@064!       jsr open               ; - call open
```

  Open up the file. This is the routine that does the real work. SETNAM and SETLFS were preparatory routines for this.

```
.@065!  ;
.@066!  ; read in the bytes and display (skipping line links etc)
.@067!  ;
.@068!        ldx #$01              ;   file #1
.@069!        jsr chkin             ; - call chkin to set input file #.
```

  Now we need to specify the input file # and tell the computer that all further chrin's are to be from file #1. (By default, it would have read from the keyboard unless we had this here).

```
.@070!        jsr chrin             ; - ignore starting address (2 bytes)
.@071!        jsr chrin
```

  Skip the starting address -- When reading the directory it is not relevant so read the bytes and discard them.

```
.@072!  skip  jsr chrin             ; - ignore pointer to next line (2 bytes)
```

  Now we skip the pointer for the next line. This is only used when loading basic programs to re-link the lines. When listing the directory they are not needed.

```
.@073!  bck1  jsr chrin
```

  This is still part of the routine that skips the pointer to the next line, yet it has a label used below that allows us to check for end of file more easily.

```
.@074!  line  jsr chrin             ; - get line # lo.
.@075!        sta temp              ; - store lo of line # @ temp
.@076!        jsr chrin             ; - get hi of line #
```

  Here we get the line # as the next 2 bytes in the file.

```
.@077!
.@078! .if computer-64              ; * if C128 then
```

  Unfortunately C= did not provide a nice routine in the KERNAL to display numeric values - however - by exploring inside the operating system a way to display numbers is there. Note that the following may look confusing -- if it does just rest assured it will print out the line # correctly.

```
.@079!        sta $61
.@080!        ldy temp
.@081!        sty $60
.@082!        lda #$00
.@083!        sta $63
.@084!        ldy temp               ;   store values for conversion.
.@085!        jsr $ba07              ; - MONITOR routine: convert to BCD values
.@086!        lda #$00
.@087!        ldx #$08
.@088!        ldy #$03
.@089!        jsr $ba5d              ; - MONITOR routine: print BCD
.@090!                               ;values in decimal
```

  This is the C128 version which uses some of the MONITOR routines to display the numeric block size.

```
.@091! .else                        ; * else if c64
.@092!        ldx temp
.@093!        jsr $bdcd             ; - print line # (w/in ROM routine).
.@094! .ife                         ; * end of computer dependant code.
```

  This is the C64 code to display a numeric value (notice how much simplified it is over the C128)...

```
.@095!
.@096!        lda #space
.@097!        jsr bsout              ; - print space
```

  Let's print a space between the filename and the block size.

```
.@098!  gtasc jsr chrin              ; - start printing filename until
.@099!                               ;end of line.
.@100!        beq chck               ;   (Zero signifies eol).
.@101!        jsr bsout              ; - Print character
.@102!        sec
.@103!        bcs gtasc              ;   and jump back.
```

  Now we start getting a character (line #98), if zero we branch out of the loop (line #100), else we display the character (#101), and jump back (#102-03).

```
.@104!  chck  lda #charret           ; - Else we need to start the next line
.@105!        jsr bsout              ;   Print a carriage return.
```

  Ah, we got to a null byte so that's the end of this line - display a car/ret.

```
.@106!        jsr chrin              ; - And get the next pointer
.@107!        bne bck1               ;   If non-zero go, strip other ptr,
.@108!                               ; and continue.
```

  This is where we branch back -- we are checking here for 2 null bytes on input. We get the first byte of the pointer and if it's non-zero then we know it's not the end of the directory so we jump back to discard the second byte at line #73.

```
.@109!        jsr chrin              ; - Else check 2nd byte of pointer
.@110!        bne line               ;   as if both 0 then = end of directory.
```

  This is a continuation of the checking above. This time we're getting the 2nd byte and checking for 0. If it's not we jump back to get and display the line # etc. If it is 0 then that means we had $0000 for the next pointer which means that it's the end of the directory.

```
.@111!  ;
.@112!  ;had 3 0's in a row so end of prog
.@113!  ;now close the file.
.@114!  ;
.@115!        lda #$01               ;   file # to close
.@116!        jsr close              ; - so close it
.@117!        jsr clrch              ; - clear all channels
.@118!        rts                    ; - and return to basic
.@119!
```

  We then close the file by specifying the file # and calling close. We then tell the computer to reset all the default input / output devices by calling clrch (remember we changed the default input channel??). And then we can return to where this routine was called from.

```
.@120! ; FILENAME string
.@121! dir    .asc "$"
```

  This is the string that is pointed to by the SETNAM call. Note that a search pattern could be set by

```
      line#121:       .asc "$hack*"
```

and by changing the length set in .A in the call in line #56.

```
.@122!
.@123! ;;---------------------------------------------------------------------
.@124!
```

```
.@125! read'err = *
.@126!
.@127! ; This routine simply grabs bytes from a channel 15 it opens up until
.@128! ;   a car/ret byte is found. Then it closes and returns.
.@129!
```

  Reading the error channel is much much more simpler than reading the directory. Basically we just open up the channel (specifying a null name) and repeatadly get bytes until a car/ret is found.

```
.@130! rderr  lda #$00               ;   length is 0
.@131!        jsr setnam             ; - call setname
```

  Setup so we don't specify a name (length = 0).

```
.@132!        lda #$0f               ;   file # (15)
.@133!        ldx #$08               ;   device # (08)
.@134!        ldy #$0f               ;   channel # (15)
.@135!        jsr setlfs             ; - set logical file #
```

  Do the equivlent of open 15,8,15.

```
.@136!        jsr open               ; - and open it.
```

  Open it.

```
.@137!  ;specify file as input
.@138!        ldx #$0f               ;   file 15 is input
.@139!        jsr chkin              ; - so specify it.
```

  Now set up file # 15 as input so we can start getting, displaying etc until a car/ret is found.

```
.@140!  ;now read in file
.@141!  loop  jsr chrin              ; - read char
.@142!        jsr bsout              ; - print char
.@143!        cmp #charret           ;   is it return?
.@144!        bne loop               ; - if not jmp back
```

  Read in and display the characters from the error channel until a char/ret is found.

```
.@145!  ;now close the file
.@146!        lda #$0f               ;   file #
.@147!        jsr close              ; - close the file
.@148!        jsr clrch              ;   restore i/o
```

  And once it is, we close the file and restore the default i/o settings.

```
.@149!  ;now return to basic
.@150!        rts
```

  And return to our caller, in this case - basic.

==========

[The Demo Corner is going to be a column where each month we'll be introduced to a new feature (some people call them bugs, we'll call them features) of the Commodore 64 or 128 in the Video and Sound areas that have commonly been shown on demos but with no mention of how to accomplish them. Note that readers may also want to take a look at the introduction to Rasters elsewhere in this magazine.]

The Demo Corner: Missing Cycles

by Pasi 'Albert' Ojala (po87553@cs.tut.fi albert@cc.tut.fi)

Written on 15-May-91 Translation 30-May-92

Missing Cycles

[all timings are in PAL, the principle applies to NTSC too]

Everybody knows that there are 63 cycles available to the C64 processor on each scan line, except for one which only provides 23 cycles (later referred to as a "bad" scan line). But what happens when we add sprites and why ?

In the C64, the VIC (video interface controller) has much more to do than just showing graphics on the screen. It also handles the memory refresh. On each scanline, it has to refresh five rows in the memory matrix and fetch fourty bytes of graphics data.

The VIC does all of this during the cycles (phase 1) that the processor is not using the memory. These cycles, however, are not sufficient when the VIC also needs to access the character and color codes for the next row. The memory bus can't be used by the CPU and the VIC at the same time, so CPU access to the bus must be denied to allow the VIC to fetch its data. Fortunately, the VIC bus (12-bit wide) allows the character (8 bits) and color (4 bits) codes to be fetched at the same time.

Understanding how sprites work

If there are sprites on the screen, the VIC needs even more cycles to fetch all of the graphics data. Scan lines are time divided so that there is enough time for all action during one line. On each line, the sprite image pointers are fetched during phase 1. If the sprite is to be displayed on that line, the three bytes of image data are fetched right after that. Out of these three fetches, two take place during phase 2 of the clock, so the processor will lose these. On average, two clock cycles are lost for each sprite that is displayed on that line.

But how is it possible for all eight sprites to only take 16-19 cycles (depending on the timing) when we have observed that one sprite requires three cycles? And why do sprites 0, 2, 4, 6 and 7 together take up as many cycles as all eight sprites ? The answer may be found in the way the VIC tells the CPU that it needs additional cycles.

The BA signal

When the VIC wants to use the bus, the BA (Bus Available) signal goes inactive. This will happen three cycles before the bus must be released ! During these three cycles, the CPU must complete all memory accesses or delay them until it has the bus again.

The CPU either completes the current instruction in the remaining cycles or sits and waits for the bus to become available again. It can't execute a new instruction as long as it doesn't have the bus. This is why cycles seem to be lost (besides those stolen directly for the sprites). Usually, all 8 sprites take 17 cycles while one sprite takes three cycles. However, the CPU may continue to execute an instruction if it does not use the bus.

Theory and speculation

Let's suppose that all the sprites are enabled and on the same scan line. Then, the VIC steals 16 cycles (2 cycles for each sprite) for the memory fetches and 3 cycles as overhead for the BA signal, for a total of 19 cycles. However, it will be usually less because the CPU will use some of the cycles when the bus request is pending.

If we now disable sprite 4, no cycles are released for the CPU's use. This is because during the previous sprite 4 data fetch, the VIC already signals that it needs the bus for the sprite 5 data fetch and BA stays low (Refer to the timing chart). Thus, the CPU never sees BA go high during sprite 4 and 2 cycles are still lost.

Accordingly, if we only turn off sprites 1, 3 and 5 we get no cycles back from the VIC. So in time-critical raster routines, always use sprites in order.

What can we do with this feature ?

How can this be useful? A good use is for synchronization. Normally, before the CPU starts to execute the raster interrupt code, it's executing an instruction of undefined cycle-length. This execution time varies from two to seven cycles.

With a sprite, you can do the synchronization with a minimal effort using a DEC or INC instruction in the right place. If the processor is early, it has to wait for the bus, otherwise it will continue to execute cycles from the instruction.

I have never experimented with any other instruction than DEC/INC, but some others should work also. You need an instruction which has a cycle that do not need the bus to be available. e.g. INC $3fff will increase the value during the fifth cycle and do not need the bus for that.

A demo program

The enclosed program includes a short raster color routine to demonstrate this strict timing and synchronization. The background color is changed 12 times on each line. The electron beam runs over eight pixels during one cycle, so the timing must be precise.

Table for PAL VIC timing for the Missing cycles

```
012345678901234567890123456789012345678901234567890123456789012 cycles

Normal scan line, 0 sprites
ggggggggggggggggggggggggggggggggggggggggrrrrr  p p p p p p p p  phi-1 VIC
                                                                 phi-2 VIC
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx phi-2 6510
63 cycles available

Normal scan line, 8 sprites
ggggggggggggggggggggggggggggggggggggggggrrrrr  pspspspspspspsps phi-1 VIC
                                                ssssssssssssssss phi-2 VIC
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxXXX                    phi-2 6510
46-49 cycles available

Normal scan line, 4 sprites
ggggggggggggggggggggggggggggggggggggggggrrrrr  psp psp psp psp  phi-1 VIC
                                                ss  ss  ss  ss   phi-2 VIC
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxXXX              xx phi-2 6510
48-51 cycles available

Bad scan line, 0 sprites
ggggggggggggggggggggggggggggggggggggggggrrrrr  p p p p p p p p  phi-1 VIC
ccccccccccccccccccccccccccccccccccccccccc                       phi-2 VIC
                                        xxxxxxxxxxxxxxxxxxxxx phi-2 6510
23 cycles available
```

```
Bad scan line, 8 sprites
ggggggggggggggggggggggggggggggggggggggggrrrrr  pspspspspspspsps phi-1 VIC
ccccccccccccccccccccccccccccccccccccccc       sssssssssssssss phi-2 VIC
                                        xxxxXXX                 phi-2 6510
4-7 cycles available


g= grafix data fetch (character images or graphics data)
r= refresh
p= sprite image pointer fetch
c= character and color CODE fetch during a bad scan line
s= sprite data fetch
x= processor executing instructions
X= processor executing an instruction, bus request pending

Observe! The left edge of the chart is not the left edge of the screen nor
         the left edge of the beam, but the sprite x-coordinate 0. If you
         have opened the borders, you know what I mean. A sprite can be
         moved left from the coordinate 0 by using x-values greater than 500.
 ____________
|  ________  |<-- Maximum sized video screen
|||        | |
|||        |<-- Normal C64 screen
|||        | |
|||________| |
||           |
||___________|
 ^ Sprite coordinate 0


--------------------------------------------------------------------------------
Demonstration program for missing cycles


COLOR0= $CE00  ; Place for color bar 0
COLOR1= $CF00  ; Place for color bar 1
RASTER= $FA    ; Line for the raster interrupt
DUMMY= $CFFF   ; Timing variable

*= $C000
        SEI             ; Disable interrupts
        LDA #$7F        ; Disable timer interrupts
        STA $DC0D
        LDA #$01        ; Enable raster interrupts
        STA $D01A
        STA $D015       ; Enable Sprite 0
        LDA #<IRQ       ; Init interrupt vector
        STA $0314
        LDA #>IRQ
        STA $0315
        LDA #$1B
        STA $D011
        LDA #RASTER     ; Set interrupt position (inc. 9th bit)
        STA $D012
        LDA #RASTER-20  ; Sprite will just reach the interrupt position
        STA $D001       ;  when it is positioned 20 lines earlier

        LDX #51
        LDY #0
        STA $D017       ; No Y-enlargement
LOOP0   LDA COL,X       ; Create color bars
        PHA
        AND #15
        STA COLOR0,X
```

```
        STA COLOR0+52,Y
        STA COLOR0+104,X
        STA COLOR0+156,Y
        PLA
        LSR
        LSR
        LSR
        LSR
        STA COLOR1,X
        STA COLOR1+52,Y
        STA COLOR1+104,X
        STA COLOR1+156,Y
        INY
        DEX
        BPL LOOP0
        CLI             ; Enable interrupts
        RTS             ; Return


IRQ     NOP             ; Wait a bit
        NOP
        NOP
        NOP
        LDY #103        ; 104 lines of colors (some of them not visible)
                        ; Reduce for NTSC, 55 ?
        INC DUMMY       ; Handles the synchronization with the help of the
        DEC DUMMY       ;  sprite and the 6-clock instructions
                        ; Add a NOP for NTSC

FIRST   LDX COLOR0,Y    ; Do the color effects
SECOND  LDA COLOR1,Y
        STA $D020
        STX $D020
        STA $D020
        STX $D020
        STA $D020
        STX $D020
        STA $D020
        STX $D020
        STA $D020
        STX $D020
        STA $D020
        STX $D020
                        ; Add a NOP for NTSC (one line = 65 cycles)
        LDA #0          ; Throw away 2 cycles (total loop = 63 cycles)
        DEY
        BPL FIRST       ; Loop for 104 lines

        STA $D020
        LDA #103        ; For subtraction
        DEC FIRST+1     ; Move the bars
        BPL OVER
        STA FIRST+1
OVER    SEC
        SBC FIRST+1
        STA SECOND+1

        LDA #1          ; Ack the raster interrupt
        STA $D019
        JMP $EA31       ; Jump to the standard irq handler

COL     BYT $09,$90,$09,$9B,$00,$99,$2B,$08,$90,$29,$8B,$08,$9C,$20,$89,$AB
        BYT $08,$9C,$2F,$80,$A9,$FB,$08,$9C,$2F,$87,$A0,$F9,$7B,$18,$0C,$6F
        BYT $07,$61,$40,$09,$6B,$48,$EC,$0F,$67,$41,$E1,$30,$09,$6B,$48,$EC
        BYT $3F,$77,$11,$11
```

```
                        ; Two color bars

--------------------------------------------------------------------------
Basic loader for Missing cycles example program (PAL)

1 S=49152
2 DEFFNH(C)=C-48+7*(C>64)
3 CH=0:READA$,A:PRINTA$:IFA$="END"THENPRINT"<clr>":SYS49152:END
4 FORF=0TO31:Q=FNH(ASC(MID$(A$,F*2+1)))*16+FNH(ASC(MID$(A$,F*2+2)))
5 CH=CH+Q:POKES,Q:S=S+1:NEXT:IFCH=ATHEN3
6 PRINT"CHECKSUM ERROR":END
100 DATA 78A97F8D0DDCA9018D1AD08D15D0A9578D1403A9C08D1503A91B8D11D0A9FA8D, 3773
101 DATA 12D0A9E68D01D0A233A0008D17D0BDACC048290F9D00CE9934CE9D68CE999CCE, 4157
102 DATA 684A4A4A4A9D00CF9934CF9D68CF999CCFC8CA10D95860EAEAEAEAA067EEFFCF, 4878
103 DATA CEFFCFBE18CEB94FCF8D20D08E20D08D20D08E20D08D20D08E20D08D20D08E20, 4403
104 DATA D08D20D08E20D08D20D08E20D0A9008810D18D20D0A967CE64C010038D64C038, 3923
105 DATA ED64C08D67C0EE19D04C31EA0990099B00992B0890298B089C2089AB089C2F80, 3483
106 DATA A9FB089C2F87A0F97B180C6F076140096B48EC0F6741E130096B48EC3F771111, 3133
200 DATA END,0

--------------------------------------------------------------------------
Uuencoded C64 executable version (PAL)

begin 644 missing.64
M`0@-"`$`4[(T.3$U,@`F"`(`EJ5(*$,ILD.K-#BJ-ZPH0[$V-"D`40@#`$-(?
MLC`ZAT$D+$$ZF4$D.HM!)+(B14Y$(J>9(I,B.IXT.3$U,CJ``(@(!`"!1K(P/
MI#,Q.E&RI4@HQBC**$$D+$:L,JHQ*2DIK#$VJJ5(*,8HRBA!)"Q&K#*J,BDI:
M*0"I"`4`0TBR0TBJ43J74RQ1.E.R4ZHQ.H(ZBT-(LD&G,P#!"`8`F2)#2$5#F
M2U-532!%4E)/4B(Z@``."60`@R`W.$$Y-T8X1#!$1$-!.3`Q.$0Q040P.$0QK
M-40P03DU-SA$,30P,T$Y0S`X1#$U,#-!.3%".$0Q,40P03E&03A$+"`S-S<SA
M`%L)90"#(#$R1#!!.44V.$0P,40P03(S,T$P,#`X1#$W1#!"1$%#0S`T.#(Y?
M,$8Y1#`P0T4Y.3,T0T4Y1#8X0T4Y.3E#0T4L(#0Q-3<`J`EF`(,@-C@T031!4
M-$$T03E$,#!#1CDY,S1#1CE$-CA#1CDY.4-#1D,X0T$Q,$0Y-3@V,$5!14%%>
M045!03`V-T5%1D9#1BP@-#@W.`#U"6<`@R!#149&0T9"13$X0T5".31&0T8X^
M1#(P1#`X13(P1#`X1#(P1#`X13(P1#`X1#(P1#`X13(P1#`X1#(P1#`X13(PH
M+"`T-#`S`$(*:`"#($0P.$0R,$0P.$4R,$0P.$0R,$0P.$4R,$0P03DP,#@XV
M,3!$,3A$,C!$,$$Y-C=#138T0S`Q,#`S.$0V-$,P,S@L(#,Y,C,`CPII`(,@^
M140V-$,P.$0V-T,P144Q.40P-$,S,45!,#DY,#`Y.4(P,#DY,D(P.#DP,CDX[
M0C`X.4,R,#@Y04(P.#E#,D8X,"P@,S0X,P#<"FH`@R!!.49",#@Y0S)&.#=!?
M,$8Y-T(Q.#!#-D8P-S8Q-#`P.39"-#A%0S!&-C<T,44Q,S`P.39"-#A%0S-&V
;-S<Q,3$Q+"`S,3,S`.@*R`"#($5.1"PP```8
``
end
size 747

--------------------------------------------------------------------------
Uuencoded C64 executable version (NTSC)

begin 644 missing.64
M`0@-"`$`4[(T.3$U,@`F"`(`EJ5(*$,ILD.K-#BJ-ZPH0[$V-"D`40@#`$-(?
MLC`ZAT$D+$$ZF4$D.HM!)+(B14Y$(J>9(I,B.IXT.3$U,CJ``(@(!`"!1K(P/
MI#,Q.E&RI4@HQBC**$$D+$:L,JHQ*2DIK#$VJJ5(*,8HRBA!)"Q&K#*J,BDI:
M*0"I"`4`0TBR0TBJ43J74RQ1.E.R4ZHQ.H(ZBT-(LD&G,P#!"`8`F2)#2$5#F
M2U-532!%4E)/4B(Z@``."60`@R`W.$$Y-T8X1#!$1$-!.3`Q.$0Q040P.$0QK
M-40P03DU-SA$,30P,T$Y0S`X1#$U,#-!.3%".$0Q,40P03E&03A$+"`S-S<SA
M`%L)90"#(#$R1#!!.44V.$0P,40P03(S,T$P,#`X1#$W1#!"1$%%0S`T.#(YA
M,$8Y1#`P0T4Y.3,T0T4Y1#8X0T4Y.3E#0T4L(#0Q-3D`J`EF`(,@-C@T031!6
M-$$T03E$,#!#1CDY,S1#1CE$-CA#1CDY.4-#1D,X0T$Q,$0Y-3@V,$5!14%%>
M045!03`S-T5%1D9#1BP@-#@S,`#U"6<`@R!#149&0T9%04)%,#!#14(Y,#!#4
M1CA$,C!$,#A%,C!$,#A$,C!$,#A%,C!$,#A$,C!$,#A%,C!$,#A$,C!$,#A%$
M+"`T-3`R`$(*:`"#(#(P1#`X1#(P1#`X13(P1#`X1#(P1#`X13(P1#!%04$Y.
M,#`X.#$P1#`X1#(P1#!!.38W0T4V-4,P,3`P,SA$-C4L(#,Y-#(`CPII`(,@R
M0S`S.$5$-C5#,#A$-CA#,$5%,3E$,#1#,S%%03`Y.3`P.3E",#`Y.3)",#@Y(
M,#(Y.$(P.#E#,C`X.4%",#@Y0RP@,S4U.`#<"FH`@R`R1C@P03E&0C`X.4,R_
M1C@W03!&.3=",3@P0S9&,#<V,30P,#DV0C0X14,P1C8W-#%%,3,P,#DV0C0XK
```

```
M14,S1C<W+"`S,C<T`"<+:P"#(#$Q,3$P,#`P,#`P,#`P,#`P,#`P,#`P,#`PO
M,#`P,#`P,#`P,#`P,#`P,#`P,#`P,#`P,#`P,#`P,#`P,#`L(#,T`#,+1
,;`"#($5.1"PP````"
``
end
size 822
```

==========

Kernal 64 / 128
by Craig Taylor (duck@pembvax1.pembroke.edu)

Introduction

When Commodore introduced the PET ages ago before the Vic-20 and Commodore 64, 128 they set in the highest memory locations a series of jumps to other routines so that users didn't need bother checking if any revisions had been made. They were assured that the address they were jumping to, would indeed, be the address that would print out a character or whatnot.

The KERNAL has grown since Commodore first introduced it, the C=128 KERNAL has fifty-seven seperate routines which are available to programmers. These routines handle functions relating to the serial devices (the bulk of them), the screen and miscellanous system routines such as scanning the keyboard, updating and reading the system clock (TI$).

Table of Routines

The following table lists the available routines, their function, address, their name, and registers affected upon exit. In addation, on the left of each line are the group that I have catagorized them under: Video(Vid), System(Sys), and Serial(Ser).

		Registers				Group		
Address	NAME	A	X	Y	F	Descritption	Vid	Sys	Ser
FF47/128	SPINSPOUT	*				Initializes I/O for fast serial			***
FF4A/128	CLOSEALL	*	*	*		Close all files on a device			***
FF4D/128	C64MODE					Switches to C=64 mode		***	
FF50/128	DMACALL	*	*			Send DMA command to REU		***	
FF53/128	BOOTCALL	*	*		*	Attempts to run boot sector		***	***
FF56/128	PHOENIX	*	*	*		Initalizes external/internal cartri.		***	
FF59/128	LKUPLA	*	*	*	*	Looks up logical device #		***	***
FF5C/128	LKUPSA	*	*	*	*	Looks up for secondary address		***	***
FF5F/128	SWAPPER	*	*	*		Switches betten 40 / 80 column screen	***		
FF62/128	DLCHAR	*	*	*		Initializes 80 column character set	***		
FF65/128	PFKEY	*	*	*	*	Installs a function key definition		***	
FF68/128	SETBNK					Sets bank for any I/O operations		***	***
FF6B/128	GETCFG	*				Get MMU configuration for a given bank		***	
FF6E/128	JSRFAR					Jumps to a subroutine in another bank		***	
FF71/128	JMPFAR					Starts executing code in another bank		***	
FF74/128	INDFET	*	*		*	Execute a LDA(fetvec),Y from a bank		***	
FF77/128	INDSTA		*		*	Stores a value indirectly in a bank		***	
FF7A/128	INDCMP		*		*	Compares a value indirectly in a bank		***	
FF7D/128	PRIMM					Outputs null-terminated string	***		***
FF81	CINT	*	*	*		Setup VIC,screen values, 8563...	***		
FF84	IOINIT	*	*	*		Initialize VIC,SID,8563,CIA for system	***	***	
FF87	RAMTAS	*	*	*		Initialize ram.		***	
FF8D	VECTOR	*		*		Reads or Writes to Kernal RAM Vectors		***	
FF90	SETMSG					Sets Kernal Messages On/Off.		***	

FF93	SECND	*		Sends secondary address after LISTN	*** ***
FF96	TKSA	*		Sends secondary address after TALK	*** ***
FF99	MEMTOP		* *	Read or set the top of system RAM.	***
FF9C	MEMBOT		* *	Read or set the bottom of system RAM.	***
FF9F	KEY			Scans Keyboard	***
FFA2	SETMO			-- Unimplemented Subroutine in All --	[N/A]
FFA5	ACPTR	*		Grabs byte from current talker	*** ***
FFA8	CIOUT	*		Output byte to current listener	*** ***
FFAB	UNTLK	*		Commands device to stop talking	*** ***
FFAE	UNLSN	*		Commands device to stop listening	*** ***
FFB1	LISTN	*		Commands device to begin listening	*** ***
FFB4	TALK	*		Commands device to begin talking	*** ***
FFB7	READSS	*		Returns I/O status byte	***
FFBA	SETLFS			Sets logical #, device #, secondary #	***
FFBD	SETNAM			Sets pointer to filename.	***
FFC0	OPEN	* * * *		Opens up a logical file.	***
FFC3	CLOSE	* * * *		Closes a logical file.	***
FFC6	CHKIN	* * * *		Set input channel	***
FFC9	CHKOUT	* * * *		Set output channel	***
FFCC	CLRCH	* *		Restore default channels	***
FFCF	BASIN	*	*	Input from channel	***
FFD2	BSOUT	*	*	Output to channel (aka CHROUT)	*** ***
FFD5	LOAD	* * * *		Load data from file	***
FFD8	SAVE	* * * *		Save data to file	***
FFDB	SETTIM			Sets internal (TI$) clock	***
FFDE	RDTIM	* * *		Reads internal (TI$) clock	***
FFE1	STOP	* *		Scans and check for STOP key	***
FFE4	GETIN	* * * *		Reads buffered data from file	***
FFE7	CLALL	* *		Close all open files and channels	***
FFEA	UDTIM	* *		Updates internal (TI$) clock	***
FFED	SCRORG	* * *		Returns current window/screen size	***
FFF0	PLOT		* * *	Read or set cursor position	***
FFF3	IOBASE		* *	Read base of I/O block	***

The Routines Themselves.

A. Error handling

For the routines in the KERNAL that return status codes (indicated by the FL status in the chart) the carry is set if there is an error. Otherwise, the carry returned is clear. If the carry is set, the error code is returned in the accumalator:

.A	Meaning
0	Stop Key pressed
1	Too Many Open Files
2	File Already Open
3	File Not Open
4	File Not Found
5	Device Not Present
6	File Was Not Opened As Input
7	File Was Not Opened As Output
8	File Name Not Present
9	Illegal Device Number
41	File Read Error

NOTE: Some of the I/O routines indicate the error code via the READST routine when setting the carry.

B. Device Numbers:

The following table lists the "standard" device numbers used by the C= Kernal.

Device #	Device Name
0	Keyboard (standard input)
1	Cassette
2	RS-232
3	Screen (standard output)
4 - 30	Serial Bus Devices
4-7	Printers (typically)
8-30	Disk Drives (typically)

C. Routine Descriptions.

Due to space limitations a fully-detailed, descriptive summary of the KERNAL routines is not feasible. However, listed below is a description of what each routine does, expected parameters and any notes on C=128/C=64 differences as well as notes to clarify any possibly confusing details.

```
Routine        : SPINSPOUT ** 128 ONLY **
 Kernal Address: $FF47
 Description   : Setup CIA for BURT protocol.
 Registers In  : .C = 0 -> SPINP (input)
                 .C = 1 -> SPOUT (output)
 Registers Out : .A destroyed
 Memory Changed: CIA, MMU.

Routine        : CLOSEALL ** 128 ONLY **
 Kernal Address: $FF4A
 Description   : Close all files on a device.
 Registers In  : .A = device # (0-31)
 Registers Out : .A, .X, .Y used.
 Memory Changed: None.

Routine        : C64MODE ** 128 ONLY **
 Kernal Address: $FF4D
 Description   : Switches to C64 Mode
 Registers In  : None.
 Registers Out : None.
 Memory Changed: -ALL- This routine initializes and calls the C64 cold start
                 routine. There is no way to switch out of C64 mode once this
                 routine is entered.

Routine        : DMACALL ** 128 ONLY **
 Kernal Address: $FF50
 Description   : Perform DMA command (for REU)
 Registers In  : .X = Bank, .Y = DMA controller command
                 NOTE: REU registers must have been previously setup.
 Registers Out : .A, .X used
 Memory Changed: Dependenant upon REU registers, REU command.

Routine        : BOOTCALL ** 128 ONLY **
 Kernal Address: $FF53
 Description   : Attempts to load and execute boot sector from a drive.
 Registers In  : .A = drive # in ascii (usually '0' / $30)
                 .X = device #
 Registers Out : .A, .X, .Y used. .C = 1 if I/O error.
 Memory Changed: As per boot sector.

Routine        : PHOENIX ** 128 ONLY **
 Kernal Address: $FF56
 Description   : Initalizes external / internatal cartridges,check for disk boot
```

```
 Registers In  : None.
 Registers Out : .A, .X, .Y used.
 Memory Changed: Calls any auto-start catridges that are installed on the system

Routine        : LKUPLA ** 128 ONLY **
 Kernal Address: $FF59
 Description   : Search file tables for a given logical device #.
 Registers In  : .A = Logical Device #.
 Registers Out : .C = 0 if found -> .A = Logical Device #,
                                    .X = Logical File #,
                                    .Y = Logical Secondary #.
                 .C =1 if not found.
 Memory Changed: None.

Routine        : LKUPSA ** 128 ONLY **
 Kernal Address: $FF5C
 Description   : Search file tables for a given secondary address.
 Registers In  : .Y = Secondary address to search for.
 Registers Out : As LKUPLA (see LKUPLA).
 Memory Changed: None.

Routine        : SWAPPER ** 128 ONLY **
 Kernal Address: $FF5F
 Description   : Switches between 40 / 80 column screen.
 Registers In  : None.
 Registers Out : .A, .X, .Y destroyed.
 Memory Changed: Screen Editor Locations.

Routine        : DLCHAR ** 128 ONLY **
 Kernal Address: $FF62
 Description   : Initializes 80 column character set.
 Registers In  : None.
 Registers Out : .A, .X, .Y destroyed.
 Memory Changed: None.

Routine        : PFKEY ** 128 ONLY **
 Kernal Address: $FF65
 Description   : Installs a function key definition
 Registers In  : .A = pointer to Z-P address (3 bytes : address lo/hi/bank.)
                 .Y = length, .X = key # (9 = Shift RUN/STOP, 10 = HELP).
 Registers Out : .C = 1 if No room, .C = 0 if successful.
                 .A, .X, .Y destroyed.
 Memory Changed: Function Key Table modified.

Routine        : SETBNK ** 128 ONLY **
 Kernal Address: $FF68
 Description   : Sets bank for any future I/O operations
 Registers In  : .A = Memory Bank, .X = Bank where filename is.
 Registers Out : None.
 Memory Changed: None.

Routine        : GETCFG ** 128 ONLY **
 Kernal Address: $FF6B
 Description   : Get MMU configuration for a given bank.
 Registers In  : None.
 Registers Out : None.
 Memory Changed:

Routine        : JSRFAR ** 128 ONLY **
 Kernal Address: $FF6E
 Description   : Jumps to a subroutine in another bank.
 Registers In  : None. (See JMPFAR for mem locations IN)
 Registers Out : None. (See JMPFAR for mem locations OUT)

Routine        : JMPFAR ** 128 ONLY **
```

```
 Kernal Address: $FF71
 Description   : Starts executing code in another bank.
 Registers In  : None.
   Memory In   :  $02 - Bank (0-15)
                  $03 - PC high
                  $04 - PC lo
                  $05 - .S (Processor status)
                  $06 - .A
                  $07 - .X
                  $08 - .Y
 Registers Out : None.
   Memory Out  : As memory in.

Routine        : INDFET ** 128 ONLY **
 Kernal Address: $FF74
 Description   : Execute a LDA(fetvec),Y from a bank.
 Registers In  : .A - pointer to Z-Page location holding address
                 .X - Bank (0-15), .Y - Index.
 Registers Out : .A = data, .X - destroyed.
 Memory Changed: None.

Routine        : INDSTA ** 128 ONLY **
 Kernal Address: $FF77
 Description   : Execute a STA(stavec),Y in a bank.
 Registers In  : .A - pointer to Z-Page location holding address
                 .X - Bank (0-15), .Y - Index.
 Registers Out : .X - Destroyed.
 Memory Changed: As per registers.

Routine        : INDCMP ** 128 ONLY **
 Kernal Address: $FF7A
 Description   : Executes a CMP(cmpvec),Y in a bank.
 Registers In  : .A = data, .X = Bank (0-15), .Y - Z-Page ptr.
 Registers Out : .X destroyed, Flags set accordingly.
 Memory Changed: None.

Routine        : PRIMM ** 128 ONLY **
 Kernal Address: $FF7D
 Description   : Prints null terminated string following JSR to current channel
 Registers In  : None.
 Registers Out : None.
 Memory Changed: Dependent upon current device.
 Example       :
               [ . . . ]
                 JSR $FF7D         ; JSR to primm, / print following string.
                 .ASC "Hi World!"  ; String to print.
                 .BYT $00          ; IMPORTANT: Null Terminated.
               [ . . . ]

 ---------------------------------------------------------------------------

Routine        : CINT
 Kernal Address: $FF81
 Description   : Setup VIC, screen values, (128: 8563)...
 Registers In  : None.
 Registers Out : None.
 Memory Changed: Screen Editor Locations.

Routine        : IOINIT
 Kernal Address: $FF84
 Description   : Initializes pertinant display and i/o devices
 Registers In  : C64: None. | C128: $0A04/bit 7
                            |          0 - Full Setup.
                            |          1 - Partial Setup. (no 8563 char)
 Registers Out : .A, .X, .Y destroyed.
```

```
 Memory Changed: CIA's, VIC, 8502 port, (C128: also optionally 8563).
 Note          : This routine automatically distinguishes a PAL system from a
                 NTSC system and sets PALCNT accordingly for use in the
                 time routines.

Routine        : RAMTAS
 Kernal Address: $FF87
 Description   : Clears Z-Page, Sets RS-232 buffers, top/bot Ram.
 Registers In  : None.
 Registers Out : .A, .X, .Y destroyed.
 Memory Changed: Z-Page, Rs-232 buffers, top/bot Ram ptrs

Routine        : VECTOR
 Kernal Address: $FF8D
 Description   : Copies / Stores KERNAL indirect RAM vectors.
 Registers In  : .C = 0 (Set KERNAL Vectors) | .C = 1 (Duplicate KERNAL vectors)
                 .XY = address of vectors    | .XY = address of user vectors
 Registers Out : .A, .Y destroyed            | .A, .Y destroyed.
 Memory Changed: KERNAL Vectors changed      | Vectors written to .XY
 Note          : This routine is rarely used, usually the vectors are directly
                 changed themselves. The vectors, in order, are :

                 C128: IRQ,BRK,NMI,OPEN,CLOSE,CHKIN,CHKOUT,CLRCH,BASIN,BSOUT
                       STOP,GETIN,CLALL,EXMON (monitor),LOAD,SAVE
                 C64 : IRQ,BRK,NMI,OPEN,CLOSE,CHKIN,CHKOUT,CLRCH,BASIN,BSOUT
                       STOP,GETIN,CLALL,USRCMD (not used),LOAD,SAVE

Routine        : SETMSG
 Kernal Address: $FF90
 Description   : Set control of KERNAL control and error messages.
 Registers In  : .A bit 7 = KERNAL Control Messages (1 = on)
                    bit 6 = KERNAL Error   Messages (1 = on)
 Registers Out : None.
 Note          : KERNAL Control messages are those defined as Loading, Found etc
                 ... KERNAL Error messages are I/O ERROR # messages which are
                 listed as follows:

Routine        : SECND
 Kernal Address: $FF93
 Description   : Sends secondary address to device after a LISTN
 Registers In  : .A = secondary address
 Registers Out : .A used.
 Memory Changed: None.
 Note          : Low level serial I/O - recommended use OPEN,CLOSE,CHROUT etc..

Routine        : TKSA
 Kernal Address: $FF96
 Description   : Sends secondary address to device after TALK
 Registers In  : .A = secondary address.
 Registers Out : .A used.
 Memory Changed: None.
 Note          : Low level serial I/O - recommended use OPEN,CLOSE,CHROUT etc..

Routine        : MEMTOP
 Kernal Address: $FF99
 Description   : Read or Set top of System Ram
 Registers In  : .C = 1 (Read MemTop)     | .C = 0 (Set MemTop)
                                          | .XY = top of memory
 Registers Out : .XY = top of memory      | None.
 Memory Changed: None.                    | Top of memory changed.
 Note          : On the C=128, this routine refers to the top of BANK 0 RAM, not
                 BANK 1 RAM.

Routine        : MEMBOT
 Kernal Address: $FF9C
```

```
 Description   : Read or Set bottom of System Ram
 Registers In  : .C = 1 (Read MemBot)     | .C = 0 (Set MemBot)
                                          | .XY = bottom of memory.
 Registers Out : .XY = bottom of memory   | None.
 Memory Changed: None.                    | Bottom of Memory changed.
 Note          : On the C=128, this routine refers to the bottom of BANK 0 RAM,
                 not, BANK 1 RAM.

Routine        : KEY
 Kernal Address: $FF9F
 Description   : Scans Keyboard
 Registers In  : None.
 Registers Out : None.
 Memory Changed: Relevant System Keyboard Values

Routine        : SETMO
 Kernal Address: $FFA2
 Description   : This is a routine who's code never made it into any versions
                 of the KERNAL on the C64, Vic-20 and C128.  Thus it is of no
                 pratical use.

Routine        : ACPTR
 Kernal Address: $FFA5
 Description   : Get byte from current talker.
 Registers In  : None.
 Registers Out : .A = data byte.
 Memory Changed: None.
 Note          : Low level serial I/O - recommended use OPEN,CLOSE,CHROUT etc..

Routine        : CIOUT
 Kernal Address: $FFA8
 Description   : Output byte to current listener.
 Registers In  : .A = byte.
 Registers Out : .A used.
 Memory Changed: None.
 Note          : Low level serial I/O - recommended use OPEN,CLOSE,CHROUT etc..

Routine        : UNTLK
 Kernal Address: $FFAB
 Description   : Commands current TALK device to stop TALKING.
 Registers In  : None.
 Registers Out : .A used.
 Memory Changed: None.
 Note          : Low level serial I/O - recommended use OPEN,CLOSE,CHROUT etc..

Routine        : UNLSN
 Kernal Address: $FFAE
 Description   : Commands current listening device to stop listening.
 Registers In  : None.
 Registers Out : .A used.
 Memory Changed: None.
 Note          : Low level serial I/O - recommended use OPEN,CLOSE,CHROUT etc..

Routine        : LISTN
 Kernal Address: $FFB1
 Description   : Commands device to begin listening.
 Registers In  : .A = device #.
 Registers Out : .A used.
 Note          : Low level serial I/O - recommended use OPEN,CLOSE,CHROUT etc..

Routine        : TALK
 Kernal Address: $FFB4
 Description   : Commands device to begin talking.
 Registers In  : .A = device #.
 Registers Out : .A used.
```

```
 Memory Changed: None.
 Note          : Low level serial I/O - recommended use OPEN,CLOSE,CHROUT etc..

Routine        : READSS
 Kernal Address: $FFB7
 Description   : Return I/O status byte.
 Registers In  : None.
 Registers Out : .A = status byte. (see section on ERROR messages).
 Memory Changed: None.

Routine        : SETLFS
 Kernal Address: $FFBA
 Description   : Set logical file #, device #, secondary # for I/O.
 Registers In  : .A = logical file #, .X = device #, .Y = secondary #
 Registers Out : None.
 Memory Changed: None.

Routine        : SETNAM
 Kernal Address: $FFBD
 Description   : Sets pointer to filename in preperation for OPEN.
 Registers In  : .A = string length, .XY = string address.
 Registers Out : None.
 Memory Changed: None.
 Note          : To specify _no_ filename specify a length of 0.

Routine        : OPEN
 Kernal Address: $FFC0
 Description   : Open up file that has been setup by SETNAM,SETLFS
 Registers In  : None.
 Registers Out : .A = error code, .X,.Y destroyed.
                 .C = 1 if error.
 Memory Changed: None.

Routine        : CLOSE
 Kernal Address: $FFC3
 Description   : Close a logical file.
 Registers In  : .A = logical file #.
 Registers Out : .A = error code, .X,.Y destroyed.
                 .C = 1 if error
 Memory Changed: None.

Routine        : CHKIN
 Kernal Address: $FFC6
 Description   : Sets input channel.
 Registers In  : .X = logical file #.
 Registers Out : .A = error code, .X,.Y destroyed.
                 .C = 1 if error
 Memory Changed: None.

Routine        : CHKOUT
 Kernal Address: $FFC9
 Description   : Sets output channel.
 Registers In  : .X = logical file #.
 Registers Out : .A = error code, .X,.Y destroyed.
                 .C = 1 if error
 Memory Changed: None.

Routine        : CLRCH
 Kernal Address: $FFCC
 Description   : Restore default input and output channels.
 Registers In  : None.
 Registers Out : .A, .X used.
 Memory Changed: None.

Routine        : BASIN
```

```
 Kernal Address: $FFCF
 Description   : Read character from current input channel.
                 Cassette - Returned one character a time from cassette buffer.
                 Rs-232   - Return one character at a time, waiting until
                            character is ready.
                 Serial   - Returned one character at time, waiting if nessc.
                 Screen   - Read from current cursor position.
                 Keyboard - Read characters as a string, then return them
                            individually upon each call until all characters
                            have been passed ($0d is the EOL).
 Registers In  : None.
 Registers Out : .A = character or error code, .C = 1 if error.
 Memory Changed: None.

Routine        : BSOUT aka CHROUT
 Kernal Address: $FFD2
 Description   : Output byte to current channel
 Registers In  : .A = Byte
 Registers Out : .C = 1 if ERROR (examine READST)
 Memory Changed: Dependent upon current device.

Routine        : LOAD
 Kernal Address: $FFD5
 Description   : Loads file into memory (setup via SETLFS,SETNAM)..
 Registers In  : .A = 0 - Load, Non-0 = Verify
                 .XY = load address (if secondary address = 0)
 Registers Out : .A = error code .C = 1 if error.
                 .XY = ending address
 Memory Changed: As per registers / data file.

Routine        : SAVE
 Kernal Address: $FFD8
 Description   : Save section of memory to a file.
 Registers In  : .A = Z-page ptr to start adress
                 .XY = end address
 Registers Out : .A = error code, .C = 1 if error.
                 .XY = used.
 Memory Changed: None.

Routine        : SETTIM
 Kernal Address: $FFDB
 Description   : Set internal clock (TI$).
 Registers In  : .AXY - Clock value in jiffies (1/60 secs).
 Registers Out : None.
 Memory Changed: Relevant system time locations set.

Routine        : RDTIM
 Kernal Address: $FFDE
 Description   : Reads internal clock (TI$)
 Registers In  : None.
 Registers Out : .AXY - Clock value in jiffies (1/60 secs).
 Memory Changed: None.

Routine        : STOP
 Kernal Address: FFE1
 Description   : Scans STOP key.
 Registers In  : None.
 Registers Out : .A = last keyboard row, .X = destroyed (if stop key)
 Memory Changed: None.
 Note          : The last keyboard row is as follows:
                 .A -> | 7   | 6   | 5   | 4   | 3   | 2   | 1  | 0
                  KEY: |STOP |Q    |C=   |SPACE|2    |CTRL |<-  |1

Routine       : GETIN
 Kernal Address: $FFE4
```

```
 Description   : Read buffered data from file.
                 Keyboard - Read from keyboard buffer, else return null ($00).
                 Rs-232   - Read from Rs-232 buffer, else null is returned.
                 Serial   - See BASIN
                 Cassette - See BASIN
                 Screen   - See BASIN
 Registers In  : None.
 Registers Out : .A = character, .C = 1 if error.
                 .XY = used.
 Memory Changed: None.

Routine       : CLALL
 Kernal Address: $FFE7
 Description   : Close all open files and channels.
 Registers In  : None.
 Registers Out : .AX used.
 Memory Changed: None.
 Note          : This routine does not _actually_ close the files, rather it
                 removes their prescense from the file tables held in memory.
                 It's recommended to use close to close files instead of using
                 this routine.


Routine        : UDTIME
 Kernal Address: $FFEA
 Description   : Update internal (TI$) clock by 1 jiffie (1/60 sec).
 Registers In  : None.
 Registers Out : .A,.X destroyed.
 Memory Changed: Relevant system time locations changed.

Routine        : SCRORG
 Kernal Address: $FFED
 Description   : Returns current window/screen size
 Registers In  : None.
 Registers Out : .X - Window Row Max
                 .Y - Window Col Max
                 .A - Screen Col Max (128 only, 64 unchanged)
 Memory Changed: None

Routine        : PLOT
 Kernal Address: $FFF0
 Description   : Read or set cursor position.
 Registers In  : .C = 1 (Read)        |      .C = 0 (Set)
                   None.              |        .X = Col
                                      |        .Y = Row
 Registers Out : .C = 1 (Read)        |      .C = 0 (Set)
                   .X = Current Col   |         None.
                   .Y = Current Row   |
 Memory Changed:  None                |      Screen Editor Locations.

Routine        : IOBASE
 Kernal Address: $FFF3
 Description   : Returns base of I/O Block
 Registers In  : None.
 Registers Out : .XY = address of I/O block ($D000)
 Memory Changed: Screen Editor Locations.
```

==========

64K VDC RAM and an alternate GEOS128 Background Screen
by Robert A. Knop Jr. (rknop@tybalt.caltech.edu, R.KNOP1 on GEnie)

I. Introduction

GEOS, both the 64 and 128 varieties, uses bitmapped screens for its output. In 40 columns, this means 8K of system memory is set aside for the main

screen. Then, in addition, GEOS also uses display buffering; in other words, GEOS allocates a second 8K as a "background" (BG) screen that is used to keep an intact copy of the foreground (FG) screen. This can be very useful for a number of reasons; one, it can be used as an undo buffer, as it is in geoPaint. When you have a delicate drawing, and then accidentally run the eraser across it, the effects of the eraser are only written to the FG screen. A click of the UNDO button brings the BG screen, with the pre-eraser version of your painting, back to the fore. Another use is for buffering the contents of the screen when something like a dialog box or a menu is written over it. When a dialog box is erased, and you see whatever had been underneatg it magically reappear, the graphics underneath are being pulled from the BG screen.

Applications have the option not to use the BG screen. Change a couple of vectors and flags, and you can use the 8K BG screen for application RAM. (This is very convenient, since the BG screen is directly above the normal 22K of application RAM in the GEOS memory map.) Of course, the application then has to provide some way of redrawing blocks of the screen hidden by menus and dialog boxes. geoWrite is an example of this; when you bring up, and exit from, a dialog box in geoWrite, there is briefly a blank rectangle on the screen before the text is redrawn on the screen.

Under GEOS128 in 80 columns, the bitmap screen is now twice as large: 640x200 instead of 320x200. The FG screen, here 16K, occupies VDC memory. The memory used for both the 40 column FG and 40 column BG screen is used for the 80 column BG screen.

GEOS128 was written for, and runs on, 128's with only the usual 16K of VDC RAM. And, it uses basically all 16K of this RAM. However, if you have 64K of VDC RAM (as is the case with 128D's, and with flat 128's that have been upgraded), you've got an additional 48K of VDC RAM that the GEOS system doesn't touch. So, why not use some of this RAM as a 80 column BG screen? Then, if you are writing an 80-column only application, you get an extra 16K, the 40-column BG screen at $6000 and the 40-column FG screen at $a000, in main memory which your application can use however it sees fit.

II. Support Routines

Only a small number of routines actually need to be written to implement this scheme; moreover, these routines are realatively straightforward. After all, we are simply copying memory from one part of VDC RAM to another. The VDC's block copy feature of the VDC is very helpful in this endeavor. (See Craig Taylor's article from last issue, or most any 128 programming guide.) The file vdc-bg.sfx, associated with this issue of the Hacking Mag, is a self-extracting archive with a number of geoProgrammer source files (in geoWrite 2.1 format) and a small dippy demonstration program. The file VDC-BG contains the following routines:

```
InitVDC       -- make sure your VDC knows it has 64K RAM
VDCImpLine    -- Imprint horizontal line from FG screen to BG screen
VDCRecLine    -- Recover horizontal line from BG screen to FG screen
VDCImpRect    -- Imprint rectangle from FG screen to BG screen
VDCRecRect    -- Recover rectangle from BG screen to FG screen
```

Each Imprint routine actually uses most of the same code as the corresponding Recover routine; all that differs is the offset to the "source" and "destination" screens in VDC RAM. (The offset for the FG screen is $0000, and for the BG screen is $4000.) The routines take the same arguments as the non-VDC Imprint and Recover routines as documented in the Hitchhiker's Guide. (You will note, however, that for whatever reason the standard GEOS ImprintLine and RecoverLine routines were only implemented for Apple GEOS.) Briefly, these are:

Routine: InitVDC

```
Pass:        Nothing

Return:      Nothing

Destroys:    a,x

Note:        This routine should be called at the very beginning of your
             program before you do any writing to the FG screen or the VDC.
```

```
Routine:     VDCImpLine
             VDCRecLine

Pass:        r3   -- left edge of line to imprint/recover (word)
             r4   -- right edge of line to imprint/recover (word)
             r11L -- y coordinate of line to imprint/recover (byte)

Return:      r3, r4 -- processed through NormalizeX

Destroys:    a,x,y,r5-r8,r11
```

```
Routine:     VDCImpRect
             VDCRecRect

Pass:        r3   -- x-coordinate of upper-left corner (word)
             r2L  -- y-coordinate of upper-left corner (byte)
             r4   -- x-coordinate of lower-right corner (word)
             r2H  -- y-coordinate of lower-right corner (byte)

Return:      r3,r4 -- processed through NormalizeX

Destroys:    a,x,y,r5-r8,r10L,r11
```

To discuss the imprint and recover line routines, consider the ASCII diagram of a portion of a line on the VDC screen. A x indicates a pixel that is in the line to be copied. The I's indicate byte boundaries; the pixel below each I is the msb of the corresponding byte in the VDC bitmap. (Bytes increase horizontally across the screen; there is no card structure found in the 80 column bitmap screen.)

```
          I       I       I       I
          ....xxxxxxxxxxxxxxxxxxxxxxxx...
               ^  \_______________/  ^
            left          I         right
          residual     full bytes   residual
```

The line moving routine needs to figure the location in VDC RAM of the leftmost full byte, the number of full bytes, and the location of the two residual bytes; additionally, it builds a bit mask for the residual bytes, with bits set corresponding to pixels to be copied. This mask is used to create the proper combination of data from the source and destination screens in the two residual bytes.

Once it knows all this, all the line routines do is (1) submit a VDC block copy to copy the full bytes, (2) read the left residual byte from the source, mask out the appropriate pixels, and OR it with the appropriate pixels from the destination, and write that one byte (3) repeat (2) for the right residual

byte.

The rectangle routines simply call the line copy routines repeatedly, (r2H)-(r2L)+1 times. (Note that while this is the most efficient way to do it from a coding time point of view <grin>, really the rectangle routines only need calculate the residual bit masks and the locations once. Thereafter, locations can be updated by adding 80 (the length of a VDC line) for each new line. The changes to the code to implement this are not difficult, and it isn't clear why I didn't make them....)

III. Use of the Routines

In a word, you use these routines whenever you would have used the normal GEOS imprint/recover routines. There are a few other consierations, though.

First of all, you need to set the flag dispBufferOn to ST_WR_FORE. The GEOS system graphic routines think that the BG screen is in main memory, and thus will not correctly use your new VDC BG screen. This means, unfortunately, that you can't just blithely go drawing graphics, assuming that they'll be buffered for recall when needed. However, it is not too much trouble to make a call to VDCImpRect either right after you've made some graphic change, or right before something potentially hazardous will happen (e.g. a call to DoDlgBox). For instance, you might have all of your main menus be Dynamic Submenus which call VDCImpRect before opening the submenu.

Second, you should load recoverVector with VDCRecRect. Since VDCRecRect takes the same parameters as RecoverRectangle, it can substitute directly for it. Once you set this vector, all menus and dialog boxes erased from the screen automatically restore the destroyed region from your VDC BG screen.

Both of these are demonstrated in the test program included in vdc-bg.sfx.

IV. Another 32K

The alert reader will have noticed that the VDC BG screen only takes as much memory as the VDC FG screen, i.e. 16K. Thus, even with this scheme, there is still another 32K of free memory in VDC RAM. Quadruple buffering, anyone?

A more tantalizing prospect would be to implement a 640x400 interlaced screen for GEOS128. This presents a number of problems, however. First, there is that terrible flicker. But, this can be made reasonable through the use of polarizing filters (in layman's terms, "sunglasses") and appropriate color choices. More seriously, the GEOS kernal graphic routines all take byte argum`>:s for Y coordinates. So, all 400 vertical pixels cannot be addressed with those routines. Thus, sombody implementing a GEOS interlaced screen is faced with re-writing all of the graphics routines. (Something to do with the 8K you've freed up at $a000, I suppose.) Since each 640x400 graphic screen would require 32K of memory for the bitmap, you could still have a VDC Background screen.

[Note: The code discussed within this article is available via anonymous FTP at tybalt.caltech.edu under the directory pub/rknop/hacking.mag as vdc-bg.sfx. This will dissolve into the GEOWRITE source files.]

==========

GeoPaint File Format
---
by Bruce Vrieling (bvrieling@undergrad.math.waterloo.edu)

GeoPaint is an excellent graphics program written for the GEOS environment. Its disk access is relatively quick, compared it to what a comparable program would do on a non-GEOS equipped C64. Part of this accomplishment can be attributed to

the diskTurbo that is an integral part of GEOS. However, the special GeoPaint file-saving scheme deserves some of the credit.

VLIR

GeoPaint files are always stored in Variable Length Indexed Recording files. VLIR files offer advantages not available without GEOS. Generally speaking, VLIR is the ultimate in RELATIVE files.

The format of a VLIR file is not that difficult to figure out. While in a regular C64 file, the two bytes directly following the FILETYPE byte in the directory would point to the data file following, VLIR files use these two bytes to point to a VLIR HEADER BLOCK (don't confuse the VLIR HEADER block with the INFO block). The first two bytes of this block are $00/FF, as the header block is a MAXIMUM of one block long. (This is why when you VALIDATE a GEOS disk from C64 mode, GeoPaint pictures are lost. Only the header block is recognised as being part of the file. The rest of the picture gets deallocated in the BAM). The remaining 254 bytes in the block are divided into 127 2-byte pointers to tracks/sectors on the disk. These pointers point to the individual records of the VLIR file, which may be ANY number of blocks long. The VLIR track/sector pointers in the VLIR header block only point to the FIRST block of the chain. From then on, the sectors chain themselves together using the normal format ie. the first two bytes of each block point to the following block.

A sample GeoPaint VLIR header might look like this:

```
0000:00 FF 03 11 03 05 03 01 ........
0008:04 03 00 FF 00 FF 00 FF ........
0010:04 07 00 00 00 00 00 00 ........
 etc....
```

The first two bytes, $00/FF, tell the drive that this is the last (and only) block in this VLIR HEADER SECTION (will never be more than 1 block, as was mentioned earlier). The next pair of bytes, $03/11, points to the first VLIR record. The next two, $03/05, point to the second record.

You will notice that 5th record contains the values $00/FF. This means that for this record, there is no picture data. We will get into exactly what the data held in the records mean in a minute. The $00/FF entries indicate an empty record. Finally, the 9th entry, $00/00 indicates the end of the GeoPaint file. There is no more data beyond this point.

One note should be made. GeoPaint is not always consistent in its handling of the data in a header block. Sometimes, it will show quite a few $00/FF combinations before finally terminating with a $00/00. When reading the file, I always read the entire header, as I don't trust the end of file method mentioned above. Just remember that any track/sector link that does not contain $00/FF or $00/00 is a valid record, with picture data.

Layout on Screen

GEOS orients the data in a GeoPaint file slightly differently than in a PhotoScrap or an Icon. A photoscrap stores the bytes corrosponding to a screen which looks like this:

```
001 002 003 004 005 ....0012
013 014 015 016 017 ....0024
```

Consecutive bytes are placed BESIDE each other. However, if you are at all familiar with the layout of a C64 hi-res screen, you will know this is very different from the layout that the VIC chip sees data. GeoPaint uses a format

identical to the VIC chip layout on screen.

GeoPaint pictures are stored in the following format:

```
001 009 017 025 033 .....  313
002 010 018 026 034 .....  314
003 011 019 027 035 .....  315
004 012 020 028 036 .....  316
005 013 021 029 037 .....  317
006 014 022 030 038 .....  318
007 015 023 031 039 .....  319
008 016 024 032 040 .....  320

321 329 .....
322 330 .....
323 331 .....
324 332 .....
325 333 .....
326 334 .....
327 335 .....
328 336 .....
```

As you can see, this is very different from the PhotoScrap format. Consecutive bytes are NOT stored on the screen beside each other. Rather, they are stored underneath each other into groups of 8 bytes. This makes moving the data from the disk onto the screen that much faster, as the decompacted bytes can just be stored on the screen after each other. Of course, this makes porting GEOS pics to the 128's VDC that much more difficult, as the VDC conforms to the PhotoScrap format.

Compression Method

GEOS uses an excellent compression method to store files on disk. You may have noticed that nearly empty pictures on disk consume very little disk space. This can be credited to GeoPaint's smart compression techniques.

Basically, the format of the compression has one COMMAND byte followed by one or more DATA bytes. The COMMAND byte tells GEOS what to do with following DATA bytes. There are 4 commands for compression:

1) If the COMMAND byte is less than 64, this indicates that there are 'COMMAND' many DATA bytes following that are to be taken as individual bytes. This is the least effective method of compression, as no compression takes place.

2) If the COMMAND byte ranges from 65 to 127, then this is a special type of compression. First of all, the next 8 bytes in the file are read in as DATA. This DATA is used to make an 8*8 'stamp'. Secondly, the amount of times to 'stamp' this 8*8 square is calculated (COMMAND AND 63). Then, the stamping is done. 'Stamping' sounds more difficult that it really is. What it boils down to, is repeating the 8 byte DATA stamp 'COMMAND AND 63' times.

3) If the COMMAND byte is 129 or greater, then the following DATA byte is repeated 'COMMAND AND 127' times. This is different from #1, as only 1 DATA byte is called in, and simply repeated. #1 called in 'COMMAND' many DATA bytes.

4) If the COMMAND byte is ZERO, we have reached the end of the VLIR record for the GeoPaint picture.

It should be noted that the COMMAND byte will NEVER be 64 or 128. If it is, there has been an error.

Format of Data After Decompacting

After the data has been decompacted, it remains to be placed on the screen. Each VLIR record holds 16 scanlines of data, or 2 character lines (different ways of looking at the same thing).

The format of the data is as follows:

> First, there is 640 bytes of picture data, comprising the first character line (8 scanlines). Remember, GeoPaint pictures are 640 pixels across. 640 pixels works out to 80 bytes. A character line is 8 pixels deep, so 80*8 comes to 640 bytes.
>
> These bytes are followed by the 640 bytes for the second chacacter line (next 8 scanlines). This is followed by 8 garbage bytes that accidentaly worked themselves into the original GeoPaint design. They should be set to zero.
>
> Finally, two sets of 80 bytes of colour data follow. The first set comprises the colour for the first line, the second 80 bytes for the second line. To wrap things up, the VLIR record is terminated by a zero byte.
>
> The next VLIR record will hold the data for the NEXT 16 scanlines, and so on.

Conclusion

That about wraps up this discussion on GeoPaint format for files. We've discussed the format of VLIR files on disk, layout of picture data on screen, compression methods used in GeoPaint files, and the format of the data once decompacted. I hope this information will come in handy for someone.

==========

Rasters - What They Are and How to Use Them
by Bruce Vrieling - (bvrieling@undergrad.math.waterloo.edu)

Anyone who has fiddled around with interrupts on the Commodore 64 has undoubtedly heard at one time or another of the concept of rasters being mentioned. Rasters are the 'ultimate' achievement of interrupt programming, or so they say. What is a raster? And how does one go about writing a program to use them?

Overview of what Interrupts are all about

A raster is sort form for the concept of a 'raster interrupt'. Before going into rasters, perhaps a brief review of interrupts is in order.

Interrupts are events generated by the CIA timer in the C64 to perform certain tasks. 60 times a second, the CIA chip signals an interrupt is due to be processed (ie. the interrupt timer timed out). This causes the 6510 CPU to stop executing the current program, save the registers on the stack, and begin to execute the interrupt code. Some of the things which get done during an interrupt include the keyboard scan, and updating TI (the software clock). When the interrupt code is finished, an RTI instruction is executed, which brings the interrupt's execution to a halt. The registers are retrieved from the stack, and the current program in memory continues to execute once again. It will continue to do so until the next interrupt occurs, about 1/60 of a second later.

The above is what happens in a normal C64 (the C128 follows the same idea, but more events occur during a C128 interrupt). [Ed. Note: In addition, the C=128

generates its interrupts via a screen raster instead of the CIA chip.]

However, you can change the normal course of events, and cause some code of your design to be added to the normal list of events which occur every interrupt. Some of the simple favourites include flashing the border 60 times per second. (Note that we have not begun the topic of rasters yet; this has nothing to do with rasters. That discussion begins at the next heading.)

How do you change the interrupt's normal course of action? It's rather simple. The C64 contains an interrupt VECTOR at locations 788/9 which is 'jumped through' before the Kernal Rom gets a chance to execute its code. If you change this vector to point to YOUR code, and make the end of your code point to the normal Kernal location (where the interrupt normally would have jumped to, $EA31), and you are careful not to step on anything, your code will be executed 60 times per second.

An example is in order:

```
; flasher
;
; this program causes the border to flash 60 times per second
;
setup = *

sei                          ; disable interrupts
lda #<intcode                ; get low byte of target routine
sta 788                      ; put into interrupt vector
lda #>intcode                ; do the same with the high byte
sta 789
cli                          ; re-enable interrupts
rts                          ; return to caller

intcode = *

inc $d020                    ; change border colour
jmp $ea31                    ; exit back to rom
```

The above is an example of a very simple interrupt routine. If you were to assemble it with an assembler, and SYS to the SETUP routine, you would see your border flash 60 times per second.

You will notice the SEI and CLI machine language instructions used above. They are very important. We don't want an interrupt occurring in between the STA 788 and the STA 789 instructions.

Think what would happen if one did: 788 would have been modified, but 789 would still be pointing to the high byte of the Kernal address. Result: the interrupt would have jumped to heaven knows where. You can be virtually guaranteed that it would NOT be pointing to a valid piece of interrupt code. Your machine would crash. The SEI instruction turns interrupts OFF, so that there is no danger of an interrupt occurring during execution of the following routine. The CLI turns them back on. If you forget to turn them back on, and accidentally leave them off, your keyboard will freeze when you return to basic, and your machine will seem to lock up.

The above was a very simple example. There are many useful things which can also be done on an interrupt. I have seen code which played music in the background of a running Basic program (it played the popular .MUS files). GEOS uses interrupts extensively to control the pointing of the mouse, and to trigger events. Interrupts are powerful beasts, and the following concept concerning raster interrupts specifically is a particularly useful animal for some people.

The Raster

A raster is a loosely used term. It refers to an interrupt that is triggered when the ray gun on the back of your monitor draws a certain line on the video screen. There are many different sources which can cause an interrupt. You are not limited to what the CIA chip can do. Rasters depend on interrupts specifically generated by the VIDEO chip. You could make this interrupt change the border colour of the screen below a certain screen line. When the screen line you specified gets redrawn, the interrupt goes off. Your code then quickly changes some memory locations to create a different video mode or effect. You could cause the bottom half of the screen to gets it's character definitions from another, different character set. Or, you could make the top 3/4 of your screen exist in hi-res multi-colour graphics, and keep the bottom 1/4 of the screen in text mode.

Some facts about the video screen: it gets redrawn exactly 60 times per second. It contains 200 scan lines on the normal 25*40 display, numbered 50 to 250 or thereabouts (note that there are more visible scan lines though: the top and bottom borders, for example). The actual re-drawing of the screen is synchronized to the electrical power coming into your house, 60 Hz. That's why some programs behave differently when run on European machines. The power is delivered at 50 Hz over there.

Why do we have to worry about a video interrupt? If the screen gets redrawn 60 times per second, and regular interrupts also occur at 60 times per second, why not simply put some code into the regular interrupt to do what we want with the screen? Because the two types of interrupts are not in sync. Neither one of them occurs EXACTLY 60 times per second, and the differences are enough to make it next to impossible to get coordinated activity of any kind happening on the screen. When we use the video interrupt, we KNOW we are at a certain line on the screen, as being on that line is what caused the interrupt to happen in the first place.

So, let's summarize. We know that regular interrupts occur 60 times per second. We also know that the video screen gets re-drawn 60 times per second, and that we can cause an interrupt to be generated when a certain line gets drawn on the screen. One slight drawback to all of this is that BOTH types of interrupts (regular and raster driven) travel through the SAME vector (ie. about 120 interrupts per second, 60 of one, and 60 of the other). Your code will have to check and see what the source of the interrupt was, and act accordingly. Or will it?

The system needs an interrupt to occur 60 times per second to do housekeeping, and uses the CIA clock to generate the interrupts. We want to interrupt every time a certain scan line is reached on the monitor, which will also just happen to occur at 60 times per second. We also have to make sure that they don't interfere with each other. The regular interrupts should be sent to their Rom destination, while our video interrupts should go to our code, and no where else.

If both are occurring at 60 times per second, why not do the job of the system Rom, and our video code on the SAME interrupt? We know that the CIA chip is not good for this; it is out of sync with the video image. Why not turn OFF the CIA interrupt, enable the raster/video interrupt, and do both jobs on one interrupt? Then we would have an interrupt signal that occurs 60 times per second, and is in perfect sync with the video image.

That's exactly what we're going to do.

Astute reads will notice a slight flaw in the above logic. For simplification purposes, I didn't get into the fact that you will need TWO raster interrupts PER SCREEN to accomplish anything useful. Why two? Because any change to the video mode you put into effect 3/4 of the way down the screen will have to be undone at the TOP of the next screen update. If you decide to make the top 3/4 of the screen a hi-res image, and the bottom 1/4 text, you need one interrupt

3/4 of the way down the screen to change from hi-res to text, but you need a SECOND one at the top of the screen to change back to hi-res from text.

So, we will now have 120 interrupts going off every second to accomplish our video desires, with 60 of them working a double shift, making sure the system interrupt code gets executed also. Remember that we are working with a specific example. There is no reason why you couldn't split the screen into N different video modes, and have (N+1)*60 interrupts going off per second. As long as you keep your code short (so your interrupts don't take too long, and have another interrupt occur before the current one is done - messy), it will work beautifully.

So far, this is all talk. Let's write a few short code segments to accomplish some of the feats we've just discussed.

The first we'll do is a re-hash of the one presented above. It flashes the border again. It does not do any mid-screen changes of video modes or anything fancy like that, so only 1 interrupt per screen is required (ie. 60 per second, not 120 etc.). This program simply shows the same idea, but this time using video interrupts as the source rather than the CIA. You probably won't notice a difference during execution.

```
------------------------------------------------------------------------
; flasher - part II
;
; this program causes the border to flash 60 times per second
; the source of the interrupts is the video chip
;
setup = *

sei                         ; disable interrupts

lda #$7f                    ; turn off the cia interrupts
sta $dc0d

lda $d01a                   ; enable raster irq
ora #$01
sta $d01a

lda $d011                   ; clear high bit of raster line
and #$7f
sta $d011

lda #100                    ; line number to go off at
sta $d012                   ; low byte of raster line

lda #>intcode               ; get low byte of target routine
sta 788                     ; put into interrupt vector
lda #<intcode               ; do the same with the high byte
sta 789
cli                         ; re-enable interrupts
rts                         ; return to caller


intcode = *

inc $d020                   ; change border colour

lda $d019                   ; clear source of interrupts
sta $d019

lda #100                    ; reset line number to go off at
sta $d012

jmp $ea31                   ; exit back to rom
```

As you can tell, there's a wee bit more to this code than there was in the original. Execute it, and you'll notice that it looks pretty much the same as the results from the first program. But there's a difference: the interrupts are now being generated from the video chip, not the CIA. For this program, it didn't make much difference. However, for a more complicated program, it makes a world of difference.

I'd better explain some of the code used above:

```
lda #$7f
sta $dc0d
```

- This piece disables any interrupts caused by the CIA chip.

```
lda $d01a
ora #$01
sta $d01a
```

- Location $d01a controls which sources may cause an interrupt (other than the normal CIA). There are 4 different possible sources: rasters, sprite to sprite collision, sprite to background collision, and the light pen. Bit #0 is the raster bit, and this piece of code activates it.

```
lda $d011
and #$7f
sta $d011
```

- This code clears bit #7 of location $d011. This location is used for many different things. Bit #7 represents the highest bit of the raster line (see segment below for more on the raster line #). More than 256 raster line numbers are possible (some are off screen, and some represent the upper and lower border areas). This bit is the 9th bit. I set it to zero because all my code affects rasters only on the normal 25*40 line display, well within the 0-255 range. This decision was an arbitrary choice on my part, to make the code simpler.

```
lda #100
sta $d012
```

- Location $d012 is the lower 8 bits of the raster line on which the interrupt is to be generated. The number 100 was another arbitrary choice. For changing border colours, the actual line number was not important. Later on, in the next example, it will become important.

```
lda #>intcode
...
rts
```

- Re-vectors the interrupt code to the new code.

```
inc $d020
```

- Changes the border colour.

```
lda $d019
sta $d019
```

- These lines clear the bit in the interrupt register which tells the source of the interrupt (in preperation for the next).

```
     lda #100
     sta $d012

     - This line resets the raster line to go off at line number 100
       again (same as above). It should be reset, so the next interrupt
       will know what line to occur on.

     jmp $ea31

     - Exit back to the Kernal Rom.
```

A Useful Example

The following is an example of a more sophisticated piece of raster code. It makes the top half of the screen border white, and the bottom half black.

```
------------------------------------------------------------------------
setup = *

; some equates

COLOUR1 = 0
COLOUR2 = 1
LINE1 = 20
LINE2 = 150

; code starts

setup = *

sei                         ; disable interrupts

lda #$7f                    ; turn off the cia interrupts
sta $dc0d

lda $d01a                   ; enable raster irq
ora #$01
sta $d01a

lda $d011                   ; clear high bit of raster line
and #$7f
sta $d011

lda #LINE1                  ; line number to go off at
sta $d012                   ; low byte of raster line

lda #>intcode               ; get low byte of target routine
sta 788                     ; put into interrupt vector
lda #<intcode               ; do the same with the high byte
sta 789

cli                         ; re-enable interrupts
rts                         ; return to caller

intcode = *

lda modeflag                ; determine whether to do top or
                            ; bottom of screen
beq mode1
jmp mode2

mode1 = *
```

```
lda #$01                    ; invert modeflag
sta modeflag

lda #COLOUR1                ; set our colour
sta $d020

lda #LINE1                  ; setup line for NEXT interrupt
sta $d012                   ; (which will activate MODE2)

lda $d019
sta $d019

jmp $ea31                   ; MODE1 exits to Rom

mode2 = *

lda #$00                    ; invert modeflag
sta modeflag

lda #COLOUR2                ; set our colour
sta $d020

lda #LINE2                  ; setup line for NEXT interrupt
sta $d012                   ; (which will activate MODE1)

lda $d019
sta $d019

pla                         ; we exit interrupt entirely.
tay                         ; since happening 120 times per
pla                         ; second, only 60 need to go to
tax                         ; hardware Rom. The other 60 simply
pla                         ; end
rti

modeflag .byte 0
```

The above code, when executed, will result in the top half of your border being white, and the bottom black. You may wish to fiddle with the equates (COLOUR1, COLOUR2, LINE1, and LINE2) to get different effects.

I see some confused faces concerning why the above exit the interrupts the way they do. Remember, since we want a split screen effect, we have to have one interrupt occur at the TOP of the screen, to turn on the WHITE effect, and one midway down to turn on the BLACK effect. Two interrupts times 60 means 120 interrupts will be executed per second. The Rom only needs 60 per second to service the keyboard and its other stuff. So, we send 60 to the Rom (the interrupts which go through MODE1) by JMPing to $EA31, and the other 60 we trash. The PLA... RTI business is the proper way to bring an interrupt to an end without going through the Rom. The RTI will ReTurn from Interrupt, and cause the regular program to continue to execute.

That brings to an end this discussion on rasters. I hope the above examples have proved to be a valuable learning tool for you. With luck, they will motivate you to continue to experiment with rasters, and come up with some neat effects.

If you have any questions, be sure to ask me about them.

==========

BURSTING YOUR 128: THE FASTLOAD BURST COMMAND

by Craig Bruce <f2rx@jupiter.sun.csd.unb.ca>

1. INTRODUCTION

This article discusses the well-unknown Fastload command of the 1571 and 1581 disk drive Burst Command Instruction Set. If you look in the back of your '71 (or '81 I presume) disk drive manual, you will find that the information given about the Fastload utility is not exactly abundant.

The Fastload command was intended to load program files into memory for execution, but it can be used just as well for reading through sequential files that would be much too large to load into a single bank of internal memory.

To make use of the Fastload burst command, I implement a word counting utility that will count the number of lines, words, and characters in a text file on a 1571 or 1581 disk drive. The advantage of using the Fastload command over regular sequential file accessing through the kernel and DOS is that the Fastload operates about 3.5 times faster on both drives.

2. WORD COUNTING UTILITY

To use the word counting program, LOAD and RUN it like a regular BASIC program. It will ask you for the name of a file. Enter the name if it is on device number 8, or put a one character prefix and a ":" if it is on another device. A "b" means device 9, "c" device 10, etc. The following are examples of valid names:

```
. filename          "filename" on device 8
. b:filename        "filename" on device 9
. a:filename        "filename" on device 8
```

The file must be on either a 1571 or 1581 disk drive; the program will not work with non-burst devices. The program will work with either PRG or SEQ files, since the Fastload command can be told not to worry about the file type.

I use the same definition of a word as the Unix "wc" command uses: a sequence of characters delimited by whitespace, where whitespace is defined to be SPACE, TAB, and NEWLINE (Carriage Return) characters. To get the line count, I simply count the number of NEWLINEs. If the last line of the file does not end with a NEWLINE character, then the count will be one line short. This is the same as the Unix wc command too. A proper text file should have its last line end with a NEWLINE character.

On my JiffyDOS-ified 1571 and 1581, I am able to achieve a word counting speed of 5,400 chars/sec and 6,670 chars/sec, respectively. I am not sure how much of a difference JiffyDOS makes, but I am not willing to rip out the ROMs to check. I tested using a 318K file.

3. BURST READ LIBRARY

This section presents the burst reading library that you can incorporate into your own programs and describes how the burst commands work. The library has three calls:

```
. burstOpen  ( .A=Device, .X=NameLen, burstBuf=Filename ) : <first block>
. burstRead  () : burstBuf, burstStatus, burstBufCount
. burstClose ()
```

I define three common storage variables for using this package: "burstBuf", "burstStatus", and "burstBufCount". "burstBuf" is a 256 byte area where the data read in from the disk drive is stored before processing, and is located at $0B00. "burstStatus" is a zero-page location that keeps the status returned from the burst command system. This is needed by the user to detect when the end of file has been encountered. "burstBufCount" gives the number of data bytes available in "burstBuf" after an open or read operation. Its

value will be somewhere between 1 and 254. A full sector contains 254 bytes of data and two bytes of control information.

"burstStatus" and "burstBufCount" are defined to be at locations $FE and $FF, respectively. You are allowed to alter the values of the two variables and the data buffer between calls, if you wish. For reasons not completely understood, interrupts must be disabled for the entire course of burst reading a file. I suspect this is because the IRQ service routine reads the interrupt mask register of CIA#1, thus clearing the SerialDataReady flag that the burst read routine waits for. Anyway, the open routine does a SEI and the close routine does a CLI, so you don't have to do this yourself.

If an error occurs during the exection of one of these routines, it will return with the carry flag set and with the error code in the .A register (same as the kernel (yes, I know that Commodore likes to call it the "kernAl")). Error codes 0 to 9 correspond to the standard kernel codes, error code 10 means that the device is not a burst device, and error codes 16 to 31 correspond to the burst controller status codes 0-15. If no error occurs, the routines return with the carry flag clear, of course.

Only one file may be open at a time for Fastloading, since Fastload takes over the disk drive and the entire serial bus. Even regular files cannot be accessed while a fastload is in progress. Thus, Fastload is not suitable for all file processing applications, but it works very well for reading a file into memory (like for a text editor) and for summarization operations (like word counting). The burst library requires that the kernel and I/O space be in context when it is called.

3.1. BURST OPEN

The way that a burst command is given is to give a magical incantation over the command channel to the disk drive. You can either use the low-level serial bus calls (LISTN, SECND, CIOUT, and UNLSN) or use the OPEN and CHROUT calls. I used the low level calls for a little extra zip.

The burst command format for Fastload is given in the back of your drive manual:

BYTE \ bit:	7	6	5	4	3	2	1	0	Value
0	0	1	0	1	0	1	0	1	"U"
1	0	0	1	1	0	0	0	0	"0"
2	P	X	X	1	1	1	1	1	159
3 - ??	<filename>								

where "X" means "don't case" and "P" means "program". If the P bit is '0' then only program (PRG) files can be loaded, and if it is '1' then sequential (SEQ) files can be loaded as well. The package automatically sets this flag for you. Note that you don't have to do an Inquire Disk or Query Disk Format in order to use this command like you have to do with the block reading and writing commands.

If you want to try giving the incantation yourself, enter:

```
OPEN1,8,15,"U0"+CHR$(159)+"FILENAME"
```

(where "FILENAME" is the name of some file that exists on your disk) on your 128 and your disk drive will spring to life and wait for you to read the file data. You can't read the data from BASIC, so to cancel the command:

```
CLOSE1
```

The "burstOpen" call of this package accepts the name of the file to be loaded at the start of the "burstBuf" buffer, the length of the filename in the .X

register, and the device number to load the file from in the .A register. The burst command header and the filename are sent to the disk drive as described above.

The open command also reads the first sector of the file, for two reasons. First, the status byte returned for the first sector has special meaning. Status code $02 means "file not found". The package translates this into the kernel error code. Second, and most important, there is a bizarre feature (read: "bug") in the Fastload command. If the file to be read is only one block long, then the number of bytes reported for the block length is two bytes too short. The following table gives the number of bytes reported and the number of actual bytes in the sector:

Actual	4	3	2	1	0
Reported	2	1	0	255	255

This is where I ran into problems with Zed-128; the logic of my program screwed up on a zero length. I have corrected the problem here, though. This bug is bizarre because it only happens if the first sector is the only sector in the file. The reported length for all subsequent sectors is correct. Note also that 255 is reported for lengths of both 1 and 0. This is because there is no actual zero length for Commodore files. If you OPEN1,8,2,"EMPTY" and then immediately CLOSE1 you get a file with one carriage return character in it.

The open routine calls the read routine to read a sector and if it was the only sector of the file, the two additional bytes are burst-read and put into the data buffer. Note that incrementing the reported count of 255 twice gives the correct count of 1. Also interesting in this case is that when the 1571/81 reports the 255, it actually transfers 255 bytes of data, all of which are bogus. It seems to me that they made the 1581 bug-compatible with the 1571 in this respect.

The open routine also executes a SEI for reasons discussed above. The information returned by the open call is the same as what is returned for the "burstRead" call discussed next.

3.2. BURST READ

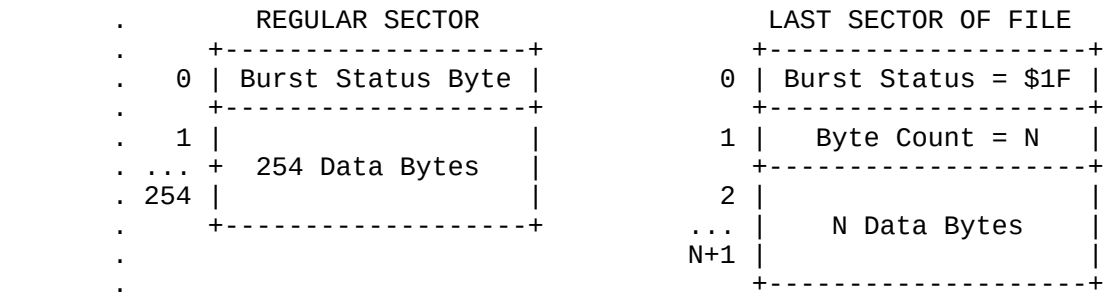
Once the Fastload command is started, the drive starts waiting to transfer the data to you. The transfer occurs sector by sector, with each sector preceeded by a burst status byte. The data is transferred using the shift register of CIA#1 and the handshaking is done using the Slow Serial Clock line of CIA#2. To receive a byte, you toggle the Slow Serial Clock line and wait for the Shift Register Ready signal from CIA#1 and then read the data value from the shift data register.

One of the clock registers in the CIA in the 1571/81 is used as the baud rate generator for the serial line. I think that it uses a delay of 4 microseconds per bit, which gives a baud rate of 250,000 (31.25K/sec). In my experimentation, the maximum baud rate I have ever achieved in reality is 200,000 (25K/sec). I read in my 1571 Internals book that the 250,000 baud rate cannot actually be achieved because of electrical problems that I don't understand. This is an important difference because the data comes flying off the surface of the disk at around 30K/sec and if the serial bus were fast enough, it could be transferred to the computer as it is being read. Some things would be so much more convenient if whomever created the universe had thought to make light go just a little bit faster.

The burst handshaking protocol slows the maximum transfer rate down to about 16K/sec. Of course, the disk drive has more things to keep on top of than just transferring data, so the actual burst throughput is lower than that: about 5.4K/sec with my JiffyDOS-ified 1571 and about 7K/sec with a 1581. Note that you can probably increase your 1571's burst performance a bit by setting

it to use a sector interleave factor of 4, using the "U0>S"+CHR$(i) burst command. By default, a 1571 writes files with an interleave of 6.

All of the sectors before the last one will contain 254 bytes of data and the last one will contain a specified number of bytes, from 1 to 254. The status code returned for the last sector is value $1F. In this case, an additional byte is sent before the data bytes that tells how many data bytes there will be. This is the value that is bugged for one-sector files as described in the last section. For those who like pictures, here are diagrams of the data transferred for a sector:



If a sector returns a burst status code other than 0 (ok) or $1F (end), then an error has occurred and the disk drive aborts the transfer, closes the burst connection, and starts the drive light blinking.

The "burstRead" call of this package reads the data of the next sector of the opened file into the "burstBuf" and returns the "burstStatus" and "burstBufCount" (bytes read). In the event of an error occuring, this routine returns with the carry flag set and the translated burst error code in the .A register (same as burstOpen). When the last sector of the file has just been read, this routine returns with a value of $1F in the "burstStatus" variable.

3.3. BURST CLOSE

After reading the last data byte of the last sector of the file, the burst connection and is closed automatically by the disk drive. The "burstClose" routine is not necessary for communication with the disk drive, but is provided for completeness and to clear the interrupt disable bit (CLI) that the open routine set to prevent interrupts while burst reading.

3.4. PACKAGE USAGE

The following pseudo-code outlines how a user program is expected to use the burst reading package:

```
.     jsr put_filename_into_burstBuf
.     ldx #filename_length
.     lda #device_number
.     jsr burstOpen
.     bcs reportError
. L1: jsr process_burstBuf_data
.     lda burstStatus
.     cmp #$1f
.     beq L2
.     jsr burstRead
.     bcc L1
.     jsr reportError
. L2: jsr burstClose
```

4. IMPLEMENTATION

This section discusses the code that implements the word counting program. It is here in a special form; each code line is preceeded by a few special characters and the line number. The special characters are there to allow you

to easily extract the assembler code from the rest of this magazine (and all of my ugly comments). On a Unix system, all you have to do is execute the following command line (substitute filenames as appropriate):

```
grep '^\.%...\!' Hack3 | sed 's/^.%...\!..//' | sed 's/.%...\!//' >wc.asm
```

```
.%001!  ;Word Count utility using the burst command set's Fastload facility
.%002!  ;written 92/06/25 by Craig Bruce for C= Hacking Net Magazine
.%003!
```

The code is written for the Buddy assembler and here are a few setup directives.

```
.%004!  .mem
.%005!  .bank 15
.%006!  .org $1c01
.%007!
.%008!  ;*** BASIC startup code
.%009!
```

This is what the "10 sys 7200" in BASIC looks like. It is here so this program can be executed with BASIC RUN command.

```
.%010!  .word $1c1c
.%011!  .word 10
.%012!  .byte $9e
.%013!  .asc  " 7200 : "
.%014!  .byte $8f
.%015!  .asc  " 6502 power!"
.%016!  .byte 0
.%017!  .word 0
.%018!  .word 0
.%019!
.%020!  jmp main
.%021!
.%022!  ;========== burst read library ==========
.%023!
.%024!  burstStatus = $fe
.%025!  burstBufCount = $ff
.%026!  burstBuf = $b00
```

"serialFlag" is used to determine whether a device is Fast or not, and the "ioStatus" (a.k.a. "ST") is to tell if a device is present or not.

```
.%027!  serialFlag = $a1c
.%028!  ioStatus = $90
```

"ciaIcr" is the interrupt control register of CIA#1. It is polled to wait for data becoming available in the shift register ("ciaData"). "ciaSerialClk" is the Slow serial bus clock line that is used for handshaking on the Fast bus.

```
.%029!  ciaIcr = $dc0d
.%030!  ciaSerialClk = $dd00
.%031!  ciaData = $dc0c
.%032!
.%033!  kernelListen = $ffb1
.%034!  kernelSecond = $ff93
.%035!  kernelCiout  = $ffa8
.%036!  kernelUnlsn  = $ffae
.%037!  kernelSpinp  = $ff47
.%038!
```

This is the error code value this package returns if it detects that a device is not Fast.

```
.%039!  errNotBurstDevice = 10
.%040!
.%041!  burstFilenameLen = burstBufCount
.%042!
.%043!  burstOpen = * ;(.A=Device, burstBuf=Filename, .X=NameLen):<first block>
```

Set up for a burst open: clear the Fast flag and the device not present flag.

```
.%044!     stx burstFilenameLen
.%045!     pha
.%046!     lda serialFlag
.%047!     and #%10111111
.%048!     sta serialFlag
.%049!     lda #0
.%050!     sta ioStatus
.%051!     pla
```

Command the disk device to Listen. Then check if the device is present or not (bit 7 of ioStatus). If not present, return the kernel error code.

```
.%052!     jsr kernelListen
.%053!     bit ioStatus
.%054!     bpl +
.%055!
.%056!     devNotPresent = *
.%057!     jsr kernelUnlsn
.%058!     lda #5
.%059!     sec
.%060!     rts
.%061!
```

Tell disk device to listen on the command channel (channel #15).

```
.%062!  +  lda #$6f
.%063!     jsr kernelSecond
```

Send the "U0"+CHR$(159) burst command header.

```
.%064!     lda #"u"
.%065!     jsr kernelCiout
.%066!     bit ioStatus
.%067!     bmi devNotPresent
.%068!     lda #"0"
.%069!     jsr kernelCiout
.%070!     lda #$9f
.%071!     jsr kernelCiout
```

Send the filename.

```
.%072!     ldy #0
.%073!  -  lda burstBuf,y
.%074!     jsr kernelCiout
.%075!     iny
.%076!     cpy burstFilenameLen
.%077!     bcc -
```

Finish sending the burst command and make sure the device is Fast.

```
.%078!     jsr kernelUnlsn
.%079!     lda serialFlag
.%080!     and #$40
.%081!     bne +
.%082!     sec
.%083!     lda #errNotBurstDevice
```

```
.%084!     rts
.%085!
```

Disable interrupts.

```
.%086!  +  sei
```

Prepare to receive data and signal the disk drive to start sending (by toggling the slow serial Clock line).

```
.%087!     clc
.%088!     jsr kernelSpinp
.%089!     bit ciaIcr
.%090!     lda ciaSerialClk
.%091!     eor #$10
.%092!     sta ciaSerialClk
```

Read the first sector of the file.

```
.%093!     jsr burstRead
```

Check for errors. Burst error code 2 (file not found) is translated to its kernel equivalent.

```
.%094!     lda burstStatus
.%095!     cmp #2
.%096!     bcc +
.%097!     bne shortFile
.%098!     sec
.%099!     lda #4
.%100!  +  rts
.%101!
```

Check if this is a one-block file.

```
.%102!     shortFile = *
.%103!     cmp #$1f
.%104!     bne openError
.%105!     ldy burstBufCount
.%106!     ldx #2
.%107!
```

If so, we have to read the two bytes that the disk drive forgot to tell us about. For each byte, we wait for for the Shift Register Ready signal, toggle the clock, and read the shift data register. I can get away with reading the data register after sending the acknowledge signal to the disk drive because I am running with interrupts disabled and it could not possibly send the next byte before I pick up the current one. We wouldn't want any NMIs happening while doing this, though.

```
.%108!     shortFileByte = *
.%109!     lda #$08
.%110!  -  bit ciaIcr
.%111!     beq -
.%112!     lda ciaSerialClk
.%113!     eor #$10
.%114!     sta ciaSerialClk
.%115!     lda ciaData
.%116!     sta burstBuf,y
.%117!     iny
.%118!     dex
.%119!     bne shortFileByte
```

Store the updated byte count and exit.

```
.%120!     sty burstBufCount
.%121!     clc
.%122!     rts
.%123!
```

In the event of a burst error, re-enable the interrupts since the user might not call the burstClose routine. Return the translated error code.

```
.%124!     openError = *
.%125!     cli
.%126!     sec
.%127!     ora #$10
.%128!     rts
.%129!
```

Read the next sector of the file.

```
.%130!  burstRead = * ;( ) : burstBuf, burstBufCount, burstStatus
```

Wait for the status byte to arrive.

```
.%131!     lda #8
.%132!  -  bit ciaIcr
.%133!     beq -
```

Toggle clock line for acknowledge.

```
.%134!     lda ciaSerialClk
.%135!     eor #$10
.%136!     sta ciaSerialClk
```

Get status byte and check. If 2 or more and not $1F, then an error has occurred. If 0, then prepare to read 254 data bytes.

```
.%137!     lda ciaData
.%138!     sta burstStatus
.%139!     ldx #254
.%140!     cmp #2
.%141!     bcc actualRead
.%142!     cmp #$1f
.%143!     bne openError
```

If status byte is $1F, then get the next byte, which tells how many data bytes are to follow.

```
.%144!     lda #8
.%145!  -  bit ciaIcr
.%146!     beq -
.%147!     ldx ciaData
.%148!     lda ciaSerialClk
.%149!     eor #$10
.%150!     sta ciaSerialClk
.%151!
.%152!     actualRead = *
.%153!     stx burstBufCount
.%154!     ldy #0
.%155!
```

Read the data bytes and put them into the burst buffer. The clock line toggle value is computed before receiving the data for a little extra zip. I haven't experimented with this, but you might be able to toggle the clock line before receiving the data (however, probably not for the first byte).

```
.%156!     readByte = *
.%157!     lda ciaSerialClk
```

```
.%158!     eor #$10
.%159!     tax
.%160!     lda #8
.%161!  -  bit ciaIcr
.%162!     beq -
.%163!     stx ciaSerialClk
.%164!     lda ciaData
.%165!     sta burstBuf,y
.%166!     iny
.%167!     cpy burstBufCount
.%168!     bne readByte
.%169!  +  clc
.%170!     rts
.%171!
```

Close the burst package: simply CLI.

```
.%172!  burstClose = *
.%173!     cli
.%174!     clc
.%175!     rts
.%176!
.%177!  ;========== main program ==========
.%178!
```

This is the word counting application code.

```
.%179!  bkWC = $0e
.%180!  bkSelect = $ff00
.%181!  kernelChrin  = $ffcf
.%182!  kernelChrout = $ffd2
.%183!
```

The "wcInWord" is a boolean variable that tells whether the file scanner is currently in a word or not. The Lines, Words, and Bytes are 24-bit counters.

```
.%184!  wcInWord = 2 ;(1)
.%185!  wcLines = 3  ;(3)
.%186!  wcWords = 6  ;(3)
.%187!  wcBytes = 9  ;(3)
.%188!
.%189!  main = *
```

Put the kernel ROM and I/O space into context then initialize the counting variables.

```
.%190!     lda #bkWC
.%191!     sta bkSelect
.%192!     jsr wcInit
```

Follow the burst reading procedure outline.

```
.%193!     jsr wcGetFilename
.%194!     jsr burstOpen
.%195!     bcc +
.%196!     jsr reportError
.%197!     rts
.%198!  /  jsr wcScanBuffer
.%199!     lda burstStatus
.%200!     cmp #$1f
.%201!     beq +
.%202!     jsr burstRead
.%203!     bcc -
.%204!     jsr reportError
.%205!  +  jsr burstClose
```

```
Report the numbers of lines, words, and characters and then exit.

.%206!     jsr wcReport
.%207!     rts
.%208!

Initialize the variables.

.%209!  wcInit = *
.%210!     lda #0
.%211!     ldx #8
.%212!  -  sta wcLines,x
.%213!     dex
.%214!     bpl -
.%215!     sta wcInWord
.%216!     rts
.%217!

Get the device and filename from the user.  Returns parameters suitable for
passing to burstOpen.

.%218!  wcGetFilename = * ;() : burstBuf=Filename, .A=Device, .X=FilenameLen

Display the prompt.

.%219!     ldx #0
.%220!  -  lda promptMsg,x
.%221!     beq +
.%222!     jsr kernelChrout
.%223!     inx
.%224!     bne -

Get the input line from the user.

.%225!  +  ldx #0
.%226!  -  jsr kernelChrin
.%227!     sta burstBuf,x
.%228!     cmp #13
.%229!     beq +
.%230!     inx
.%231!     bne -
.%232!  +  jsr kernelChrout

Extract the device number from the start of the input line.  If it is not
there, assume device number 8.

.%233!     lda #8
.%234!     cpx #2
.%235!     bcc filenameExit
.%236!     ldy burstBuf+1
.%237!     cpy #":"
.%238!     bne filenameExit
.%239!     sec
.%240!     lda burstBuf
.%241!     sbc #"a"-8
.%242!     tay

If a device name was present, then we have to move the rest of the filename
back over it now that we've extracted it.

.%243!     ldx #0
.%244!  -  lda burstBuf+2,x
.%245!     sta burstBuf,x
.%246!     cmp #13
```

```
.%247!     beq +
.%248!     inx
.%249!     bne -
.%250!  +  tya
.%251!     filenameExit = *
.%252!     rts
.%253!
.%254!     promptMsg = *
.%255!     .asc "enter filename in form filename, or a:filename, "
.%256!     .asc "or b:filename, ..."
.%257!     .byte 13
.%258!     .asc "where 'a' is for device 8, 'b' is for device 9, ..."
.%259!     .byte 13,0
.%260!
```

Scan the burst buffer after reading a sector into it.

```
.%261!  wcScanBuffer = *
.%262!     ldy #0
.%263!     cpy burstBufCount
.%264!     bne +
.%265!     rts
.%266!  +  ldx wcInWord
.%267!  -  lda burstBuf,y
.%268!  ;   jsr kernelChrout  ;uncomment this line to echo the data read
.%269!     cmp #13
.%270!     bne +
```

If the current character is a carriage return, then increment the line count.

```
.%271!     inc wcLines
.%272!     bne +
.%273!     inc wcLines+1
.%274!     bne +
.%275!     inc wcLines+2
```

If the character is a TAB, SPACE, or a RETURN, then it is a Delimiter;
otherwise, it is considered a Letter.

```
.%276!  +  cmp #33
.%277!     bcs isLetter
.%278!     cmp #" "
.%279!     beq isDelimiter
.%280!     cmp #13
.%281!     beq isDelimiter
.%282!     cmp #9
.%283!     beq isDelimiter
.%284!
.%285!     isLetter = *
```

If the character is a Letter and the previous one was a Delimiter, then
increment the word count.

```
.%286!     cpx #1
.%287!     beq scanCont
.%288!     ldx #1
.%289!     inc wcWords
.%290!     bne scanCont
.%291!     inc wcWords+1
.%292!     bne scanCont
.%293!     inc wcWords+2
.%294!     jmp scanCont
.%295!
.%296!     isDelimiter = *
.%297!     ldx #0
```

```
.%298!
.%299!     scanCont = *
.%300!     iny
.%301!     cpy burstBufCount
.%302!     bcc -
```

Add the number of bytes in the burst buffer to the total byte count for the file.

```
.%303!     clc
.%304!     lda wcBytes
.%305!     adc burstBufCount
.%306!     sta wcBytes
.%307!     bcc +
.%308!     inc wcBytes+1
.%309!     bne +
.%310!     inc wcBytes+2
.%311!  +  stx wcInWord
.%312!     rts
.%313!
```

Report the number of lines, words, and bytes read. Uses a "printf" type of scheme.

```
.%314!  wcReport = *
.%315!     ldx #0
.%316!  -  lda reportMsg,x
.%317!     beq reportExit
.%318!     cmp #13
.%319!     bcs +
.%320!     stx 14
.%321!     tax
.%322!     lda 2,x
.%323!     sta 15
.%324!     lda 0,x
.%325!     ldy 1,x
.%326!     ldx 15
.%327!     jsr putnum
.%328!     ldx 14
.%329!     jmp reportCont
.%330!  +  jsr kernelChrout
.%331!     reportCont = *
.%332!     inx
.%333!     bne -
.%334!     reportExit = *
.%335!     rts
.%336!
.%337!     reportMsg = *
.%338!     .byte 13
.%339!     .asc "lines="
.%340!     .byte wcLines
.%341!     .asc ", words="
.%342!     .byte wcWords
.%343!     .asc ", chars="
.%344!     .byte wcBytes,27
.%345!     .asc "q"
.%346!     .byte 13,0
.%347!
```

Reports the error number given in the .A register. Called after an error is returned from a burst routine.

```
.%348!  reportError = * ;( .A=errNum )
.%349!     pha
.%350!     ldx #0
```

```
.%351!  -  lda errorMsg,x
.%352!     beq +
.%353!     jsr kernelChrout
.%354!     inx
.%355!     bne -
.%356!  +  pla
.%357!     ldy #0
.%358!     ldx #0
.%359!     jsr putnum
.%360!     lda #13
.%361!     jsr kernelChrout
.%362!     rts
.%363!
.%364!     errorMsg = *
.%365!     .asc "*** i/o error #"
.%366!     .byte 0
.%367!
.%368!  ;==========library==========
.%369!
```

Routine to print out the 24-bit number given in .AYX.

```
.%370!  libwork = $60
.%371!  itoaBin = libwork
.%372!  itoaBcd = libwork+3
.%373!  itoaFlag = libwork+7
.%374!
.%375!  putnum = *
```

Initialize binary and BCD (Binary Coded Decimal) representations of number.

```
.%376!     sta itoaBin+0
.%377!     sty itoaBin+1
.%378!     stx itoaBin+2
.%379!     ldx #3
.%380!     lda #0
.%381!  -  sta itoaBcd,x
.%382!     dex
.%383!     bpl -
.%384!     sta itoaFlag
.%385!     ldy #24
.%386!     sed
.%387!
```

Rotate each bit out of the binary number and then multiply the BCD number by two and add the bit in. Effectively, we are shifting the bits out of the binary number and into the BCD representation of the number.

```
.%388!     itoaNextBit = *
.%389!     asl itoaBin+0
.%390!     rol itoaBin+1
.%391!     rol itoaBin+2
.%392!     ldx #3
.%393!  -  lda itoaBcd,x
.%394!     adc itoaBcd,x
.%395!     sta itoaBcd,x
.%396!     dex
.%397!     bpl -
.%398!     dey
.%399!     bne itoaNextBit
.%400!     cld
```

Take the BCD bytes and spit out the two digits they contain.

```
.%401!     ldx #0
```

```
.%402!     ldy #0
.%403!  -  lda itoaBcd,x
.%404!     jsr itoaPutHex
.%405!     inx
.%406!     cpx #4
.%407!     bcc -
.%408!     rts
.%409!
.%410!     itoaPutHex = *
.%411!     pha
.%412!     lsr
.%413!     lsr
.%414!     lsr
.%415!     lsr
.%416!     jsr itoaPutDigit
.%417!     pla
.%418!     and #$0f
.%419!
```

Print out the individual digits of the number. If the current digit is zero and all digits so far have been zero, then don't output anything, unless it is the last digit of the number.

```
.%420!     itoaPutDigit = *
.%421!     cmp itoaFlag
.%422!     bne +
.%423!     cpy #7
.%424!     bcc itoaPutDigitExit
.%425!  +  ora #$30
.%426!     sta itoaFlag
.%427!     jsr kernelChrout
.%428!     itoaPutDigitExit = *
.%429!     iny
.%430!     rts
```

5. UUENCODED PROGRAM

Here is the binary executable in uuencoded form. The CRC32 of it is 3676144922. LOAD and RUN it like a regular BASIC program.

```
begin 640 wc
M`1P<'`H`GB`W,C`P(#H@CR`V-3`R(%!/5T52(0``````3!$=AO](K1P**;^-
M'`JI`(60:""Q_R20$`<@KO^I!3A@J6\@D_^I52"H_R20,.NI,""H_ZF?(*C_
MH`"Y``L@J/_(Q/^0]2"N_ZT<"BE`T`0XJ0I@>!@@1_\L#=RM`-U)$(T`W2"]
M'*7^R0*0!=`$.*D$8,D?T"&D_Z("J0@L#=SP^ZT`W4D0C0#=K0S<F0`+R,K0
MYX3_&&!8.`D08*D(+`W<\/NM`-U)$(T`W:T,W(7^HO[)`I`6R1_0W:D(+`W<
M\/NN#-RM`-U)$(T`W8;_H`"M`-U)$*JI""P-W/#[C@#=K0S<F0`+R,3_T.48
M8%@88*D.C0#_(#T=($D=(",<D`0@H!Y@(`4>I?[)'_`((+T<D/(@H!X@#AT@
M6QY@J0"B")4#RA#[A0)@H@"]C1WP!B#2_^C0]:(`(,__G0`+R0WP`^C0\R#2
MLZD(X`*0'JP!"\`ZT!<XK0`+Z3FHH@"]`@N=``O)#?`#Z-#SF&!%3E1%4B!&
M24Q%3D%-12!)3B!&3U)-($9)3$5.04U%+"!/4B!!.D9)3$5.04U%+"!/4B!"
M.D9)3$5.04U%+"`N+BX-5TA%4D4@)T$G($E3($9/4B!$159)0T4@."P@)T(G
M($E3($9/4B!$159)0T4@.2P@+BXN#0"@`,3_T`%@I@*Y``O)#=`*Y@/0!N8$
MT`+F!<DAL`S)(/`;R0WP%\D)\!/@`?`1H@'F!M`+Y@?0!^8(3$0>H@#(Q/^0
MQ1BE"67_A0F0!N8*T`+F"X8"8*(`O8(>\!_)#;`5A@ZJM0*%#[4`M`&F#R#,
M'J8.3'X>(-+_Z-#<8`U,24Y%4ST#+"!73U)$4ST&+"!#2$%24ST)&U$-`$BB
M`+V\'O`&(-+_Z-#U:*``H@`@S!ZI#2#2_V`J*BH@22]/($524D]2(",`A6"$
M889BH@.I`)5CRA#[A6>@&/@&8"9A)F*B`[5C=6.58\H0]XC0[-BB`*``M6,@
D!!_HX`20]F!(2DI*2B`/'V@I#\5GT`3`!Y`'"3"%9R#2_\A@````
`
end
```

6. REFERENCES

[1] Commodore Business Machines, _Commodore_1571_Disk_Drive_User's_Guide_,

CBM, 1985.

[2] Rainer Ellinger, _1571_Internals_, Abacus Software, June 1986.

==========

Next Issue: (hopefully!)

Learning ML - Part 4

In the next issue we'll embark on a project of making a space invaders style game for the C=64/128 using the KERNAL routines we've learned.

The Demo Corner: FLI - more color to the screen

All of us have heard complaints about the color constraints on C64. FLI picture can have all of the 16 colors in one char position. What then is this FLI and how it is done ?

The 1351 Mouse Demystified

An indepth look at how the 1351 mouse operates and how to access it within your own ML programs. For Basic programmers, a driver for the 80 column screen is also supplied.

LITTLE RED READER: MS-DOS file reader for the 128 and 1571/81 drives.

This article will present a package that reads MS-DOS files and the root directory of MS-DOS disks. This package will use the dynamic memory allocation package introduced in Hacking Issue #2 to allow large files to be read in. The application-level code hasn't been finalized yet, but it will probably use a menu-oriented full-screen display and will read and translate MS-DOS and Commodore files.

==========

END of Commodore Hacking Issue 3.

==========